# Event-Condition-Action Rule Languages over Semistructured Data

**George Papamarkos**

March 2007

A Dissertation Submitted to
Birkbeck College, University of London
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

School of Computer Science & Information Systems
Birkbeck College
University of London

To my parents and my partner Ioanna

# Acknowledgments

First and foremost, my gratitude goes out to my two supervisors, Prof. Alexandra Poulovassilis and Dr. Peter Wood. They have been, each one in their own way, real mentors that deeply influenced my way of thinking and working. I will always be grateful for their encouragement, continuous support and indefatigable guidance.

My thanks also to many colleagues and friends at the School of Computer Science & Information Systems and London Knowledge Lab: Rajesh Pampapathi, Lucas Zamboulis, Dionisis Dimakopoulos, Hao Fan and especially Aristoklis Anastasiadis for their precious friendship and timely help during my years at Birkbeck. I cannot forget to mention George Roussos for his help and his advice at some critical times during my research.

Special appreciation also goes to my good friend and flatmate Yiannis Papantoniou for all the time we have spent together during these years and our intense and profound conversations under the depressing skies of Hyperborea. I cannot forget my friend and flatmate Chrissy Kotretsou and my childhood friend Elias Karavangelis for their everlasting jocular attitude, even during difficult periods for us all.

Last, but not least, my utmost gratitude must go to my parents, to whom I will be forever indebted for their love, support, wise guidance and their dedication throughout the years, without which I would have never be in the position to write this thesis. I thank also my brother Demosthenes, not only for his axiomatic love and support, but also for all those moments we have spent on arguing during our sparkling and passionate conversations. Finally, I owe a deep appreciation to my partner Ioanna for her inexhaustible patience, devotion and for pouring oil on my, often troubled, waters during these four years when we started together this long and unpredictable journey.

# Abstract

The increasing use of semistructured data technologies such as XML and RDF, in dynamic applications in areas such as data integration and e-commerce make it necessary to support reactivity over such data. Event-Condition-Action (ECA) rules are one of the technologies that can provide such functionality.

In this thesis we investigate potential languages and supportive technologies that enable reactive functionality over both XML and RDF data using ECA rules.

We first review the syntax and the execution semantics of an ECA language for XML data and conduct a study regarding its expressiveness. We also describe an implementation of this language that can be utilised to provide reactivity over XML files and we study its performance using analytical and experimental methods.

We introduce a new language for supporting ECA rules over RDF data, called RDFTL, and define its semantics. We also describe the architecture and implementation of a system that provides reactive functionality over RDF data in a peer-to-peer environment using RDFTL. Issues relating to indexing and management of RDF data, indexing and management of RDFTL rules, and change detection are discussed.

We develop an analytical model for analysing the behaviour of our RDFTL rule processing system using as the main performance criterion the time required to complete all rule processing after an initial top-level update. For the purposes

of the performance study, we employ both this analytical model and a simulation of the system.

The overall conclusions of this thesis are that ECA rules are a promising technology for providing reactive functionality over the commonly used semistructured data formats today, XML and RDF, in both centralised and distributed environments. In the case of peer-to-peer environments, choosing the appropriate network topology can enable the scalability of ECA rule processing and points to the practical usefulness of ECA rules as a means to provide reactivity in such environments.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Event-Condition-Actions Rules and the Web

An *Event-Condition-Action (ECA)* rule performs *actions* in response to *events* provided stated *conditions* hold. For example, in a database system an event may be any update operation on the data, e.g. inserting a new tuple into a table, and may lead to an action which is another update operation, e.g. the deletion of another tuple in another table. This behaviour, whereby actions are performed in response to events, is described as *reactive behaviour*.

ECA rules are used to enable this kind of reactive behaviour in many settings including active databases [Pat99, WC96], workflow management [CFP99], publish/subscribe technology [BCP01a, BCP01b, ABEYH00] and implementing business processes [AVFY98, IO01]. ECA rules specify the desired reactive behaviour and can be manually defined by users or generated by applications. In their general form, they consist of three parts:

- *The event part* describes the events that the rule should respond to. When a rule responds to an event then the rule is said to be *triggered*.

- *The condition part* describes a condition that has to hold in order for the rule to *fire*.

- *The action part* describes the actions to be performed by the rule if the specified event has occurred and the condition has evaluated to True.

The fact that ECA rules are defined and managed within a single rule base rather than being encoded in diverse programs, and the high-level declarative syntax of ECA rules making them amenable to analysis and optimisation techniques, are two key advantages of using ECA rules to support reactive functionality in applications as opposed to implementing such functionality directly using a programming language.

The increasing number of dynamic Web applications, the rise of Semantic Web [TJO01] technologies together with the increasing adoption of distributed architectures and peer-to-peer technologies set the scene of today's application development and deployment.

The Semantic Web is not a separate Web but an extension of the current Web, in which Web-based information is assigned a description in a controlled formalism, better enabling computers and people to work in cooperation. The Semantic Web is based on XML [W3C06a] and RDF [W3C04c] as its fundamental standards for storing and exchanging information in Web applications. This, combined with the increasing number of dynamic applications requiring timely notification of data changes, makes timely this study of ECA rules as a candidate technology for providing reactive functionality over XML and RDF data.

A peer-to-peer (P2P) network is a network of computers in which all the participants, named *peers*, share a part of their resources in order to provide services and content offered by the network as a whole. The peers act as both resource (service and content) providers and resource requestors. One classification of P2P networks is according to their degree of centralisation. In *pure* P2P networks [YGM01], peers are equals with no central peer or group of peers managing the network. All peers provide all or some of the overall set of services and can establish connection with any other peer in the network. In *hybrid* P2P networks [YGM01] there is a peer (or group of peers), often called *superpeer*(s), that generally provide more services and resources than other peers, and control the way other peers join and leave the network and are connected, and the way that messages are routed through the network.

## 1.2 Motivation for the Research

Our work on the design and the implementation of an ECA rule language over RDF data was motivated by the "SeLeNe: Self e-Learning Networks" project [SeLeNe]. The aim of this project was to investigate techniques for managing evolving RDF repositories of educational metadata and for providing a wide variety of services over such repositories, including syndication, notification and personalisation services. According to the SeLeNe User Requirements Document [KPPL03], peers in a SeLeNe (Self e-Learning Network) store RDF descriptions relating to learning objects registered with the SeLeNe, and also RDF descriptions relating to users of the SeLeNe. Each peer manages some fragment of the overall RDF/S descriptions.

SeLeNe's reactive functionality was intended to provide the following aspects of the user requirements [PPW03a]:

- automatic notification to users of the registration of new LOs of interest to them;

- automatic notification to users of the registration of new users who have information in common with them in their personal profile;
- automatic notification to users of changes in the description of resources of interest to them;
- automatic propagation of changes in the description of one resource to the descriptions of other, related resources, e.g. propagating changes in the description of a LO to the description of any composite LOs defined in terms of it.

In the SeLeNe project, we identified and investigated ECA rules over RDF data as a candidate technology for providing this reactive functionality.

## 1.3 Problem Statement

The research problems considered in this thesis are the following:

- What constructs and features should be present in ECA languages in order to support the definition and processing of ECA rules over XML and RDF data?
- What are the architectural requirements of systems that support ECA rule processing over XML and RDF data in centralised and P2P environments?
- What factors affect the performance of such ECA rule processing systems?
- What are the performance trends and the scalability characteristics of such ECA rule processing systems?

This new combination of ECA rules and semi-structured data gives rise to new research issues, including the design of new architectures to support the processing of ECA rules over centralised or distributed XML and RDF repositories, the design of ECA rule languages for XML and RDF data, event detection and rule execution over such data and the development of analysis and optimisation techniques for these new languages.

Although ECA rules are supported by most of the modern relational database systems, there is currently limited support for ECA rules over semi-structured data. Triggers on XML data are supported by all the major DBMS vendors and also by some native XML repository vendors [eXist, Xindice, dbXML]. However, these are confined to document-level triggering, and only events concerning insertion, deletion or update of an entire XML document can be detected, thus limiting the ability to define rules that trigger in response to data modification within the stored XML documents.

The complexity of XML and RDF compared to the relational data model, in conjunction with the relative immaturity of their supporting technologies and standards make research on the above issues a challenging task. The specific research problems addressed in this thesis are listed next.

## 1.4 Dissertation Outline

The outline of this thesis is as follows:

Chapter 2 gives a review of semi-structured data models and languages, including a review of query and update languages that may be used to define the different parts of ECA rules (i.e. the event, condition and action part) and a review of existing ECA rule languages for semi-structured data.

Chapter 3 reviews a particular ECA language for XML data, called XTL, including its syntax and execution semantics, investigating also the language's expressiveness. The chapter concludes by comparing XTL with another similar language for XML ECA rules (Active XQuery [BBCC02]), with respect to their syntax and semantics.

Chapter 4 describes a prototype system that supports the definition and processing of XTL rules, in a centralised environment, including its architecture and its major components. A performance study of the system is conducted, including system modelling using analytical methods and experiments conducted using both the analytical model and the system itself. An indexing structure for XTL rules is also proposed and an analytical study of its effect on the system's performance is presented.

Chapter 5 describes our ECA language for RDF, called RDFTL, including the syntax and denotational semantics of its query and update sublanguages of RDFTL rules.

Chapter 6 describes a prototype system that supports the definition and processing of RDFTL rules in P2P environments, including the system architecture and its major components. We give details of rule registration and rule execution, as well as possible approaches for concurrency control and recovery in such environments.

Chapter 7 studies the performance and scalability of our RDFTL rule processing system. We develop a performance model based on analytical methods and we use the experimental results from both the analytical model and system simulations in order to investigate the system's performance.

Finally, Chapter 8 gives our conclusions and directions of future work.

## 1.5 Thesis Contributions

We investigate how ECA rule processing systems over semistructured data can be built, in both centralised and in P2P environments, starting from the design of the ECA language to the design and implementation of the supporting rule processing systems and also a study of the systems' performance. We have designed the first ECA rule language for RDF data and defined its denotational and execution semantics.We have also designed and implemented the first system supporting RDF ECA rule processing in P2P environments. This is also the first time that a performance study over such a system has been conducted, using both analytical and simulation methods.

# Chapter 2

# Semi-Structured Data Models and Languages

## 2.1 Introduction

A data model is considered to be *semi-structured* if it is self-describing, in other words, schema information is contained within the data, or if the schema places only loose constraints on the data [Bun97]. This has a bearing on the design of ECA rule systems over semi-structured data because there is a wider variety of granularities of data updates than with structured data (for example, fragments of XML documents and subgraphs of RDF graphs). This results in a wider variety of event types and in more complex event types and more complex event detection logic. We discuss the specific design issues in more detail in later Chapters (Chapter 3, Section 3.2 and Chapter 5, Sections 5.1 and 5.2).

This chapter reviews ECA languages for semi-structured data. Because ECA languages use query and update languages to define the different parts of ECA rules, we also conduct a review of query and update languages for semi-structured data. Since the dominant semi-structured data standards today are XML and

RDF we first focus on the definition of these data models. We describe the structure of each data model, followed by the supporting schema definition languages for XML and RDF data. A review of query and update languages for XML and RDF, including the existing standard languages or the various proposals, follows. A discussion of proposals for ECA languages for semi-structured data concludes the review.

The chapter is organised as follows. In Section 2.2 we present the XML and RDF data models, including the corresponding schema definition standards for each data model, DTD [W3C06a], XMLSchema [W3C06d] and RDFSchema [W3C04b]. In Section 2.3 we review the main query languages for semi-structured data, including XML and RDF. Section 2.4 considers update languages for XML and RDF data. Finally, in Section 2.5 we review languages used for defining ECA rules both for structured and semi-structured data.

## 2.2 Semi-Structured Data Models

### 2.2.1 The XML Data Model

The *eXtensible Markup Language (XML)* [W3C06a] appeared as a W3C standard recommendation in February 1998. The development of XML was motivated by the earlier *Standard Generalized Markup Language* (SGML), and XML was originally designed to meet the challenges of large-scale electronic publishing, motivated by the inflexibility of HTML and the complexity of SGML. The ability to impose rules regarding the vocabulary and the structure of an XML document, using languages such as DTD or XMLSchema, as well as specifying document elements that are *mandatory* and *optional*, is one of the advantages of the XML standard over its HTML predecessor.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shares>
    <share name="INTL">
        <day-info day="03" month="03">
            <prices><price time="09:05">125.25</price>
                <price time="09:00">123.55</price>
                <price time="09:10">123.00</price>
            </prices>
            <high>123.55</high><low>123.00</low>
        </day-info>
        <day-info day="04" month="03">
            <prices><price time="09:05">128.25</price>
                <price time="09:00">124.55</price>
                <price time="09:10">123.00</price>
            </prices>
            <high>124.55</high><low>123.00</low>
        </day-info>
        <month-info month="03"><high>124.55</high>
                     <low>123.00</low><month-info>
    </share>
    <share name="MDRK">
        <day-info day="03" month="04">
            <prices><price time="10:10">120.55</price>
                    <price time="11:20">123.55</price>
            </prices>
            <high>123.55</high><low>120.55</low>
        </day-info>
        <day-info day="05" month="04">
            <prices><price time="10:10">123.55</price>
                <price time="10:20">124.66</price>
            </prices>
            <high>124.55</high><low>121.55</low>
        </day-info>
        <month-info month="04"><high>124.55</high>
                     <low>121.55</low><month-info>
    </share>
</shares>
```

Figure 2.1: Example XML document, `shares.xml`

Figure 2.1 shows an XML document that holds information regarding the prices of shares per day. An XML document can be seen as either a tree or as a linear document. An XML tree is required to have a *root* node under which all the other nodes appear. An XML tree consists of six different types of nodes: elements indicated by a tag name; attributes; text nodes; processing instructions; comments; and the root node. The names of the element tags and attributes are case sensitive and must begin with a *letter*, *underscore* or *colon* followed by any number of additional *letters*, *digits*, *colons*, *underscores*, *hyphens* and *periods*. In the linear representation of an XML tree, each opening element tag, e.g. <`tag-name`>, is required to be paired with its the corresponding ending tag, e.g. <`/tag-name`>. A shorthand allowed for empty elements, e.g. `<price/>` for `<price></price>`. Elements need to be correctly nested, in the sense that if a start tag `A` precedes a start tag `B` then the end tag `B` must precede the end tag `A`. The values of attributes need to be enclosed in quotes, either single or double e.g. `name='INTL'`. The first line in the XML document is the so-called *XML declaration* and defines the XML version and, optionally, the character encoding used in the document, e.g. `<?xml version="1.0" encoding="ISO-8859-1"?>`. Tags beginning with `<?` and ending with `?>` indicate *processing instructions*, while all characters enclosed within `<!--` and `--!>` are considered comments. Finally, the `CDATA` (character data) sections are enclosed in `<![CDATA[` and `]]>` and their contents are not interpreted during XML document parsing.

### 2.2.2 Schema Definition Languages for XML

DTD [W3C06a] and XML Schema [W3C06d] are the two main schema definition languages for XML. DTD (Document Type Definition) provides facilities for defining the names and the structure of XML elements, the attributes of elements, and the default values of attributes, if any. Figure 2.2 shows a DTD corresponding to the structure of the XML document shown in Figure 2.1.

```
<!ELEMENT shares (share*) >
<!ELEMENT share (day-info*)>
<!ELEMENT day-info (prices,high,low)>
<!ELEMENT month-info (high,low)>
<!ELEMENT prices (price+)>
<!ELEMENT high (#PCDATA)>
<!ELEMENT low (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST share
          name CDATA #REQUIRED>
<!ATTLIST day-info
          day CDATA #REQUIRED
          month CDATA #REQUIRED>
<!ATTLIST month-info
          month CDATA #REQUIRED>
<!ATTLIST price
          time CDATA #REQUIRED>
```

Figure 2.2: Example DTD document, `shares.dtd`

`ELEMENT` and `ATTLIST` are the two main syntactical components for defining the element structure and the attributes associated with elements, respectively. `ELEMENT` is followed by the name of the element and its content specification. The content of an element is specified using a notation similar to regular expressions and it defines the list of elements the current element may comprise. For example, in the example DTD, the `shares` element comprises of zero or more `share` elements, while `day-info` comprises of one or more `prices` elements followed by `high` and `low` elements. An element may also comprise of a text element indicated by `#PCDATA`, e.g. element `high`. Other content indicators include `EMPTY`, which defines an element with empty content, and `ANY` indicating an element that can contain any content. Elements with *mixed content* are also allowed, which may comprise of combinations of elements and text.

`ATTLIST` specifies the attributes of an element. Each attribute definition comprises an *attribute name* e.g. `time`, a *type* e.g. `CDATA` and a *default* that can be `#IMPLIED`, meaning that the attribute can omitted, `#REQUIRED` meaning that the attribute must be present, `#FIXED x` meaning that its value must be `x`, or simply `x` specifying the default value of the attribute. A DTD also has the ability to declare entities. Entities are a physical unit such as a character, a string or a file. Their declaration associates a name with an entity, e.g. `<!ENTITY myname "George Papamarkos">`. We can refer to an entity within an XML document by prefixing its name with `&`, e.g. `&myname;`.

A DTD can be specified either internally, i.e. within the XML document, or externally, in another file. The `<!DOCTYPE>` construct specifies the DTD to be used in an XML document. For example, in our example XML document in Figure 2.1, adding `<!DOCTYPE shares SYSTEM "shares.dtd">` after the `<?xml ...?>` instruction states that `shares.dtd` is the DTD to which the root element `<shares>` of the document conforms.

DTDs suffer from a number of limitations, the most significant of which are: (a) lack of support for data typing, especially for element content, (b) non-XML syntax, (c) unable to enforce order and number of child elements of mixed content elements, (d) the elements of a DTD are global, (e) difficulty of enforcing the presence of child elements without also enforcing the order and (f) no easy support for namespaces.

XMLSchema [W3C06d] overcomes these limitations. XMLSchema employs an XML syntax and supports data types for both elements and attributes. There are two kinds of data types: *simple*, that contain only text, and *complex*, that can contain elements or attributes. The simple data types include `string, boolean, date, float, decimal` and many more (see [FW04] for a full list). It is also possible to define complex data types as a combination of simple data types.

Figure 2.3 shows an XML Schema describing the XML document of Figure 2.1. The `<shares>`, `<share>`, `<day-info>` and `<prices>` elements are represented by complex types containing either other nested complex types or simple type elements or attributes, such as the `<high>` element under the `<day-info>` which holds the highest prices of the day and is of type `xsd:float`.

### 2.2.3 The RDF Data Model

The Resource Description Framework (RDF) [W3C04e] is the language proposed by the W3C to support the Semantic Web initiative by allowing the definition of metadata describing web-based resources. RDF is one of the older specifications of the W3C, with the first working draft produced in 1997. The first recommended RDF specification, including the RDF Model and Syntax, was released in 1999 with the RDF Schema Specification following in 2000.

The basic structural unit of RDF is the *triple*. Each triple consists of a *subject*, a *predicate* (also called a *property*) and an *object*. Graphically, a triple can be represented by a node representing the subject, a node representing the object, and a directed arc from the former to the latter corresponding to predicate. A set of triples thus constitutes an *RDF Graph*. An RDF triple states that some relationship, indicated by the predicate, holds between the entities denoted by subject and object of the triple. The meaning of an RDF graph is the logical conjunction of the statements corresponding to all the triples it contains. A node may be a URI (Uniform Resource Identifier [BLCG92]) or a literal, which uniquely identify the node, or may be *blank* and have no identifier. Literals are used to identify entities such as numbers and dates by means of a lexical representation. A literal may be the object of an RDF triple, but not the subject or the predicate.

There is no built-in concept of data types in RDF and the predefined XML Schema data types are used for this purpose. XML Schema capabilities can also

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="shares" type="WholeShares"/>
    <xsd:complexType name="WholeShares" >
        <xsd:sequence>
            <xsd:element name="share" type="ShareInfo" minOccurs="0"
                                               maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ShareInfo">
        <xsd:sequence>
             <xsd:element name="day-info" type="DayInfo" minOccurs="0"
                                               maxOccurs="unbounded"/>
             <xsd:element name="month-info" type="MonthInfo" minOccurs="0"
                                               maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="DayInfo">
        <xsd:sequence>
            <xsd:element name="prices" type="DayPrices"
                                               maxOccurs="unbounded"/>
            <xsd:element name="high" type="xsd:float"/>
            <xsd:element name="low" type="xsd:float"/>
        </xsd:sequence>
        <xsd:attribute name="day" type="xsd:integer"/>
        <xsd:attribute name="month" type="xsd:integer"/>
    </xsd:complexType>
    <xsd:complexType name="DayPrices">
        <xsd:sequence>
            <xsd:element name="price" type="xsd:float"
                                               maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="time" type="xsd:time"/>
    </xsd:complexType>
    <xsd:complexType name="MonthInfo">
        <xsd:sequence>
            <xsd:element name="high" type="xsd:float"/>
            <xsd:element name="low" type="xsd:float"/>
        </xsd:sequence>
        <xsd:attribute name="month" type="xsd:integer"/>
    </xsd:complexType>
</xsd:schema>
```

Figure 2.3: Example XMLSchema, `shares.xsd`

be used for defining new data types for use in RDF. XML content is also a possible value for RDF literals. Such content is indicated in an RDF graph using a literal whose data type is the built-in data type `rdf:XMLLiteral`.

To illustrate, the RDF graph in Figure 2.4 describes the metadata of a learning object. The oval nodes indicate RDF resources and the rectangles RDF literals. The nodes are connected via arcs labelled by a property name. Each property name consists of two parts separated by a ":". The left part is a reference to the namespace to which the property belongs, e.g. `dc` for Dublin Core (see Section 2.2.4 below) and `rdf` for the core RDF namespace, and the right part is the name of the property within the specified namespace. The properties in the `rdf` namespace named `_1`,`_2`, `_3` etc. indicate members of an RDF *collection*. An RDF collection can either be a sequence (order of its members matters), a bag (order-independent) or a set of alternative options. In an RDF graph, a collection is represented by a blank node with an outgoing arc labelled `rdf:type`, pointing to a resource that indicates the type of the collection, which can be `rdf:Seq` for a sequence, `rdf:Bag` for a bag or `rdf:Alt` for a set of alternatives. The contents of a collection are represented as objects of the triple having the blank node as its subject and an arc labelled `rdf:_i`, where `i` is an integer indicating the index of the member within the collection.

In addition to the abstract RDF syntax defined by the RDF graph, RDF has a recommended XML serialization, RDF/XML [W3C04c], which can be used to encode RDF data for exchange of information between applications.

### 2.2.4 RDFSchema

RDFSchema [W3C04b] is a vocabulary description language for RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources. RDF Schema employs a class and property system similar

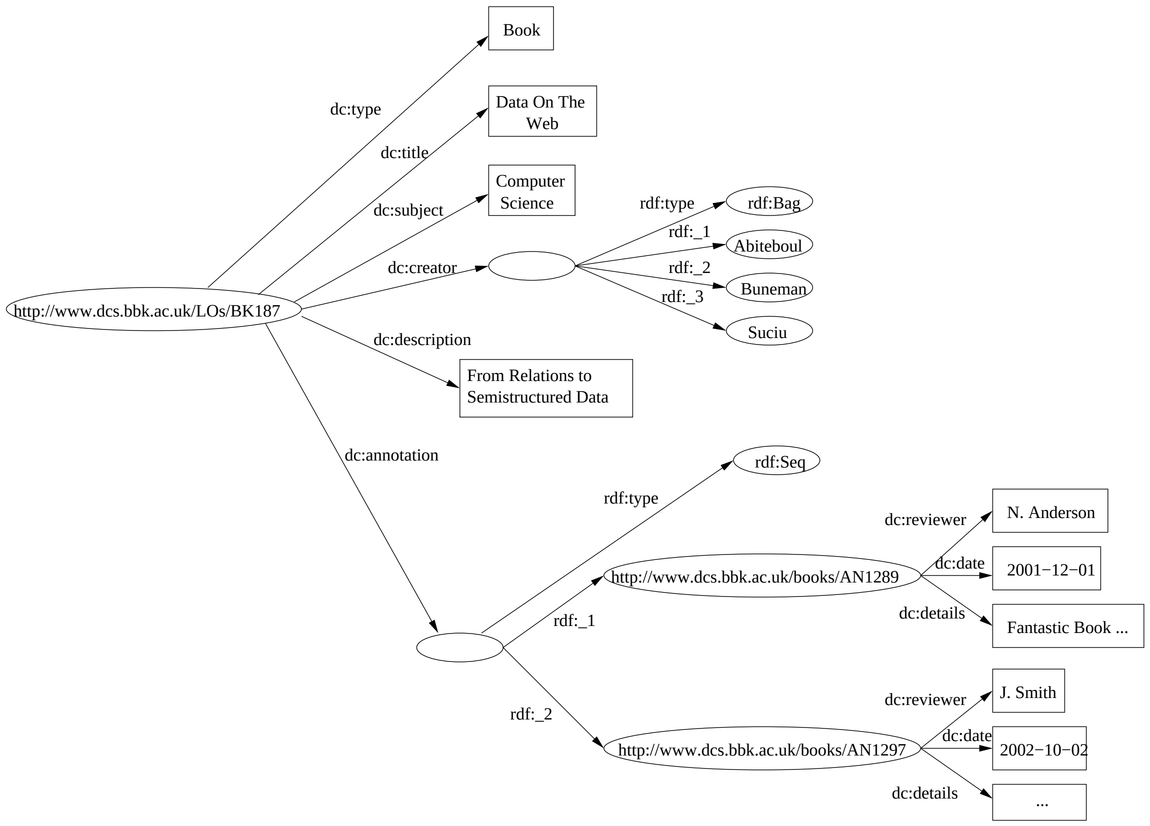


Figure 2.4: RDF data describing a learning object.

to the type system of object-oriented programming languages. The language provides a set of RDF resources that can be used to describe properties of other RDF resources and properties. The core vocabulary is defined in a namespace called `rdfs` (recall that the prefix `rdf` is used to refer to the RDF namespace). Users are free to create their own vocabularies, to extend the `rdfs` and `rdf` namespaces, and to create new namespaces.

The graph illustrated in Figure 2.5 is an example RDF Schema that the RDF data shown in Figure 2.4 conforms to. Figure 2.5 uses the Dublin Core [DC03] RDFS vocabulary, indicated by the `dc` prefix and the core RDF vocabulary indicated by the `rdf` prefix. The Dublin Core Element Set RDFS vocabulary is designed to describe identification information for a broad range of data resources, such as their title, author, type etc.

RDF resources may be divided into classes the members of which are termed instances of the class. Classes are themselves resources that can be identified by a URI. In order to state that a resource is an instance of a class, the `rdf:type` property is used, with the instance resource as the subject and the class resource as the object. The primitive RDF Schema class `rdfs:Class` is an instance of itself.

RDF Schema allows class extension via subclassing. The `rdfs:subClassOf` property may be used to state that one class is a subclass of another. If a class *C* is a subclass of a class *C'*, then all instances of *C* will also be deemed to be instances of *C'*. The following list enumerates the basic classes used in the RDF Schema vocabulary (see [W3C04b] for the full RDF Schema vocabulary):

- `rdfs:Resource`: The class that all the RDF resources are instances of. It is an instance of `rdfs:Class`.

- `rdfs:Literal`: The class of literal values such as strings and integers. It is an instance of `rdfs:Class` and a subclass of `rdfs:Resource`.

- `rdfs:Datatype`: The class of data types. It is an instance of `rdfs:Class` and a subclass of `rdfs:Literal`.

- `rdf:XMLLiteral`: The class representing XML literal values. It is an instance of `rdfs:Datatype` and a subclass of `rdfs:Literal`.

- `rdf:Property`: The class of RDF properties. It is an instance of `rdfs:Class` while `rdfs:range`, `rdfs:domain` and `rdf:type` are instances of `rdfs:Property`. `rdfs:range` is used to define the class or classes the values of a property are instances of while `rdfs:domain` is used to state that any resource that has a given property is an instance of one or more classes.

In Figure 2.5, the oval nodes represent RDF classes that are subclasses of `rdfs:Resource` and the rectangular nodes represent literals that are instances of `rdfs:Datatype`. Although not shown in Figure 2.5, the relationship between `rdfs:Resource` and an RDF class can be represented by outgoing arc, labelled `rdfs:subClassOf`, from the subject class to `rdfs:Resource`. In a similar way, an outgoing arc labelled `rdf:type` from a literal to `rdfs:Datatype` indicates that a node is an instance of `rdfs:Datatype`. For brevity, `String` and `Date` appearing in the literal nodes in the figure indicate the data type of the literal nodes whereas the formal representation would be a triple from the literal to the appropriate RDF class representing the data type. Finally, consider property $p$ with domain $d$ and range $r$. In our RDF schema representation shown in Figure 2.5, rather than this being represented by a pair of triples $(p, \texttt{rdfs}:\texttt{domain}, d)$ and $(p, \texttt{rdfs}:\texttt{range}, r)$, we instead use the single triple $(d, p, r)$. We use this representation for brevity, and it is similar to the representation used by RQL (see Section 2.3.2) for performing queries.

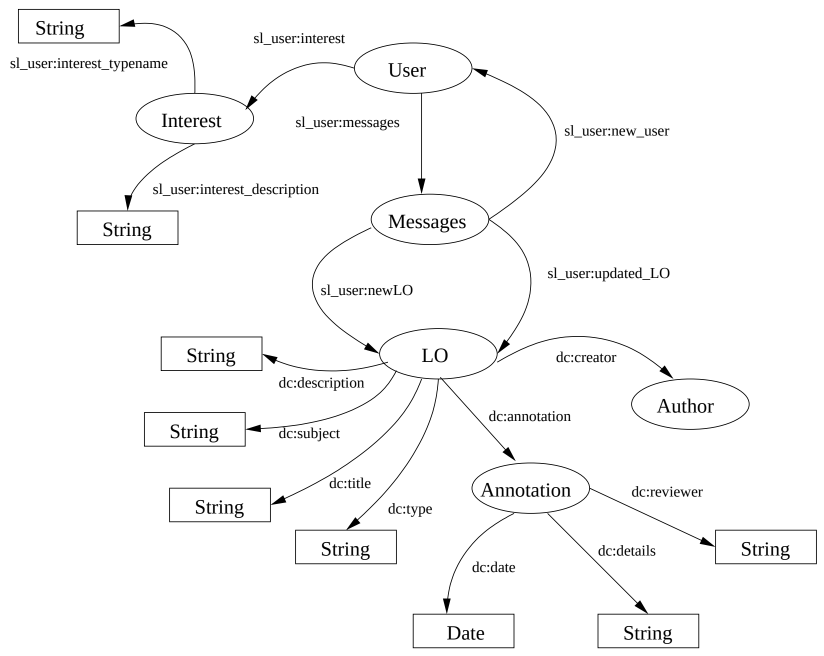


Figure 2.5: RDF schema describing users and learning objects.

## 2.3 Query Languages

We now review the main query language proposals for XML and RDF data. In the case of XML we concentrate on the two standardised query languages XPath [W3C05a] and XQuery [W3C05b], while for RDF, due to lack of any query language standardisation, we review the major query languages that have been proposed for RDF to date.

### 2.3.1 XML Query Languages

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to query XML data sources has become increasingly important.

XPath 1.0 [W3C99a] is a declarative language for specifying paths in trees, such as XML documents, using a path syntax similar to the one used in filesystem directory and file hierarchies. It was proposed by a W3C recommendation in 1999 and was designed in order to provide a means to address XML elements from within the XPointer language [W3C01] or to match sets of elements from within the XSLT language [W3C99b]. XPointer is a language used for addressing the internal structure of XML documents, providing specific references to elements, strings and other parts of XML documents. XSLT is language that specifies rules for transforming an XML document to another document, that can either be another XML document, an HTML document or a document comprising some formatting rules.

XPath treats the XML document as a *tree* of nodes, distinguishing six types of nodes. The root of the tree (which is different from, and the parent of, the document's root element); other elements; attributes; text; comments; and processing instructions. Each XPath expression is either an *absolute* or a *relative* expression. An absolute expression starts with a `'/'` followed by a relative expression and is evaluated starting at the root node. The relative expression is a sequence of location steps. Each relative path expression is evaluated with respect to an initial *context*, corresponding to a set of nodes which is defined externally by XQuery, XSLT or XPointer language constructs. So each location step is evaluated with respect to a context and produces a new set of nodes that provides the context for the next location step.

XPath employs two syntax formats, the abbreviated and the full syntax. In this thesis we mostly employ the abbreviated syntax and we refer the reader to [W3C99a] for details of the full syntax.

In the abbreviated XPath syntax, a location step can either be empty, i.e. `//`, or can specify an element name, or can specify an attribute name prefixed by `@`. An empty step identifies all descendants of all nodes in the current context. An element name identifies all child elements of each node in the context that have the given name. An attribute name identifies the attribute node (if any) of each node in the context that has the given name. Optionally, an element name and an attribute name may be followed by a *predicate*, enclosed in `[ ]`, to filter the evaluated nodes. Predicates comprise boolean expressions using `and, or, not, =`, numerical expressions, other path expressions, or built-in XPath functions.

For example, consider the XML document in Figure 2.1. The XPath expression `/shares/share` evaluates to all the `share` elements of the document, as does the expression `//share`. The expression `//share/day-info[high][low]` evaluates to all `day-info` nodes that have both `high` and `low` nodes as children, while `//share/@name` returns all the `share` nodes that have have an attribute with name `name`. To illustrate the use of predicates, `//share[@name="INTL"]` evaluates to the shares with name equal to `INTL` and
`//share/day-info[@month="04"][high=124.55]`
evaluates to all `day-info` nodes that are for April and whose highest share value of the day is `124.55`.

Similar to filesystem navigation path expressions, XPath allows the use of `'.'` and `'..'` to refer to a current context node or to the parent node of a current context node. Using `'*'` we can refer to element nodes with any name. Using `[i]`, where `i` is an integer, we can refer to the $i^{th}$ element of a set of nodes. Again considering the XML document in Figure 2.1, `//share[2]` evaluates to the second share node, i.e. the one with `name="MDRK"`, while
`//high[.=124.55]/..[@month="03"]/prices/*` evaluates to all child nodes of the `prices` node of the `day-info` nodes of March that have `high` node equal to `124.55`.

XPath lacks features such as the ability to perform joins between XML documents, projections of XML data nor result set transformation operations. XQuery is a W3C proposal for a more expressive query language for XML. Its syntax is based on the earlier QUILT [RCF00] query language, while it has also borrowed features from XQL [IKK98], SQL and OQL [OQL01] and, especially, XPath. It is a functional language that consists of expressions, which can possibly be nested. The path expressions of XQuery are based on the abbreviated syntax of XPath. XQuery employs a type system based on XML Schema and shares a set of built-in functions and operators with XPath.

The *FLWR* (pronounced "flower") expressions of XQuery support iteration and binding of variables to intermediate result sets using the `for` and `let` keywords. This kind of expression is useful for computing joins between two or more documents and for restructuring XML data. The `for` clause contains one or more variables each associated with an expression. The value of this expression is called the *binding sequence*. The `for` clause is used to iterate over the items in the binding sequence, binding the variable to each item in turn. A `let` clause may also contain one or more variables, each with an associated expression. Unlike a `for` clause, however, a `let` clause binds each variable to the entire result set of its associated expression. The optional `where` clause serves as a filter for the tuples of variable bindings generated by the `for` and `let` clauses. The expression in the `where` clause is evaluated once for each of these tuples and if the result is True, then the tuple is retained and its variable bindings are used in the execution of the `return` clause, or otherwise the tuple is discarded. The `return` clause is evaluated once for each tuple output by the `where` clause and the results of these evaluations are concatenated to form the result of the FLWR expression. The `order by` clause, if any, contains one or more ordering specifications according to which the returned results will be sorted.

For example, the following XQuery expression selects and returns the highest price of each day for the share named `INTL`, in ascending order of these highest prices:

```
for $x in document("shares.xml")/shares/share
let $day := $x/day-info
where $x/@name="INTL"
order by $day/high
return $day/high
```

Here the `for` clause instantiates the variable `$x` with each `share` element under the `shares` element. The `let` clause holds a reference to all the `day-info` elements of any `share`. The `where` clause selects only `share` elements with `name` attribute equal to `INTL`. The `order by` clause sorts those according to the value of their `high` elements. The `return` clause specifies the elements to be returned, in this case the `high` elements. The query will return:

```
<high>123.55</high>
<high>124.55</high>
```

XQuery allows conditional expressions to be used within any of the expressions in the `for`, `let` or `where` clauses. It also allows constructs that can create any XML structure within the `return` clause— including elements, attributes, text, comments and processing instruction nodes. For example, the following XQuery expression is similar to the previous one except that it returns the `high` element of each day in March within a new `march` element and the `high` element of each day of other months within a `rest-of-the-year` set of element tags:

```
for $x in document("shares.xml")/shares/share
let $day := $x/day-info
where $x/@name="INTL"
return if ($day/@month="03")
           then <march>$day/high</march>
           else <rest-of-the-year>$day/high</rest-of-the-year>
```

The query will return:

```
<march>123.55</march>
<march>124.55</march>
```

### 2.3.2 RDF Query Languages

Unlike XML, no standard for the RDF query language has yet emerged, although several such languages have been proposed, some inspired by the SQL and OQL query languages and some others that have a rule-based syntax. We review here the most notable proposals. All our example queries refer to the RDF graph shown in Figure 2.4 and schema in Figure 2.5.

RQL [KAC+02] is a typed functional language. It combines querying of schema and data. It is supported by in the RDFSuite system [RDFSuite] and is also partially supported by Sesame system [Sesame]. RQL supports path expressions allowing variables on both nodes and edges of the RDF graph. For example, the following query on the RDF Schema in Figure 2.5 returns the classes that appear as domain and range of the property `dc:creator`:

```
SELECT $C1, $C2
FROM {$C1}dc:creator{$C2}
USING NAMESPACE dc = &http://purl.org/dc/elements/1.1/
```

The `USING NAMESPACE` clause defines the namespaces used in the query. At the RDF data level, the following RQL query on the RDF graph illustrated in Figure 2.4 returns the title of the Learning Object resource with the specified URI:

```
SELECT Y
FROM {X}dc:title{Y}
WHERE X = &http://www.dcs.bbk.ac.uk/LOs/BK187
USING NAMESPACE dc = &http://purl.org/dc/elements/1.1/
```

RDQL [W3C04d] was the query language supported by the Jena RDF management framework [Jena2] up to version 2.2. It has an SQL-like syntax for selecting parts of the RDF graph, also allowing the use of namespace abbreviations. It is not able to query RDF Schema elements. The following example query returns all the RDF resources that are of type "`Book`":

```
SELECT ?x
WHERE (?x, dc:type, "Book")
USING dc FOR <http://purl.org/dc/elements/1.1/>
```

The `USING ... FOR` clause is used to define the necessary namespaces.

SPARQL [W3C06c] has replaced RDQL in the Jena framework since version 2.3. It too has an SQL-like select syntax for selecting RDF subgraphs, and also allows namespace abbreviations within queries by using the `PREFIX` keyword. It has been initially developed as part of the *SPARQL Protocol for RDF* [W3C06b]. This protocol was developed by the W3C RDF Data Access Working Group with the aim of standardising the submission of SPARQL queries from query clients to query processors in a distributed query environment. The SPARQL query below selects the titles of all learning objects that are of type "`Book`":

```
PREFIX dc : http://purl.org/dc/elements/1.1/
SELECT ?bookTitle
WHERE
  { ?y  dc:type  "Book" .
    ?y  dc:title  ?bookTitle .
  }
```

Versa [Versa] has been developed as part of the 4Suite [4Suite] toolset for XML and RDF data. The main aim of Versa is to create a language that can be easily integrated with the existing XML processing technologies such as XSLT. As part of this, Versa has been primarily motivated by XPath. A Versa expression can consist of either a function call, a literal, a variable reference or a nested expression. For example, the query

```
all()-dc:reviewer->*
```

applied on the RDF data shown in Figure 2.4, returns the objects of triples with subject any existing resource and property `dc:reviewer`, i.e. `N. Anderson` and `J. Smith`.

## 2.4 Update Languages

Update languages enable a higher level modification of data than via API calls from within programming language code. In this section we review the main update language proposals for XML and RDF data.

### 2.4.1 Update Languages for XML

In recent years, various languages for updating XML data have been proposed, and there are two main approaches: (a) extending SQL with XML update features and (b) XML-specific update languages.

Most commercial relational database systems, such as Oracle XML DB [OraXML], DB2 XML Extender [DB2XML] and Microsoft SQL Server 2005 [MSSQL05], adopt the first approach, providing their XML extensions with a proprietary API for updating XML data using an SQL-like syntax to express the update.

One of the earliest proposals regarding updating XML data using an XML-specific update language is presented in [TIHW01], where extensions to XQuery are presented that support a set of update operations for XML data, along with algorithms for implementing these update operations within XML-enabled relational databases. The update operations include: (a) *deletion* of an XML node,

(b) *rename* of a non-text node, (c) *insertion* of a new XML subtree or text node *before* or *after* a specified position in a list of siblings, (d) *replacement* of an XML node by another and (e) *sub-updating* that starting from a given element, uses a pattern-matching operation over the input document, possibly filtered by a condition, and the update operation is invoked recursively for each matching.

XUpdate [LM00] was proposed by the XML:DB working group [XML:DB] as a language for updating XML documents. XUpdate is expressed as a well-formed XML document, using a notation similar to XSLT, and making use of XPath to select parts of the XML document for updating. However, the XUpdate project seems to have ceased, since no progress has been reported since September 2000.

Another XML update language, defined as part of our own XML ECA language, is presented in [BPW02a] and is described later, in Chapter 3 of this thesis. It supports INSERT and DELETE actions over XML data and employs a fragment of XPath to select the XML data to be modified and a fragment of XQuery to construct new XML data fragments. DELETE actions use the XPath fragment to select the XML data to be deleted. INSERT actions can use XPath expressions to specify the insertion of an existing XML fragment under an XML node, and can be the XQuery fragment to construct a new XML subtree to be inserted under an existing node.

A recent working draft of the W3C [CFR05] adopts and extends the approach of [TIHW01], aiming to extend XQuery with an update facility. This extension supports the following operations (the examples below refer to the XML document in Figure 2.1):

- Insertion of a node. An `insert` expression inserts copies of one or more nodes into a designated position e.g. to insert a new `price` element under the price list of the last `day-info` node of the first `share` from the `shares` list:

```
do insert <price time="12:45">154.34</price>
after doc("shares.xml")/shares/share[1]/day-info[last()]
                                   /prices/price[last()]
```

- Deletion of a node. A `delete` expression deletes one or more nodes, e.g. to delete the last `day-info` element from the first `share` from the `shares` list:

```
do delete doc("shares.xml")/shares/share[1]/day-info[last()]
```

- Modification of a node by changing some of its properties while preserving its identity. The modification operations include `replace`, to replace a node by another and `rename` to rename the tag name of an element. For example, to replace the first `day-info` node of the first `share` by a copy of its last `day-info`:

```
do replace doc("shares.xml")/shares/share[1]/day-info[1]
with doc("shares.xml")/shares/share[1]/day-info[last()]
```

  and to rename all `high` nodes as `highest-price`.

```
do rename doc("shares.xml")/shares/share/day-info/high
as "highest-price"
```

- Creation of a modified copy of a node with a new identity, using the `transform` operation, e.g.

```
for $e in //share[name = "INTL"]
return
   transform
      copy $je := $e
```

```
        modify do delete $je/day-info[@month="03"]
        return $je
```

Here, the `transform` expression creates a copy `$je` of the `$e` variable, that matches all `shares` with name `INTL`, and then deletes all the daily information for March from all the shares within the `$je` copy before returning the modified copy.

As implied by the last example, FLWR syntax can be extended with these update expressions. This is also the case for conditional expressions, e.g. if the name of the first `share` in the list of `shares` is `INTL`, then the `high` element of the first `day-info` of the first `share` can be replaced by a copy of the `high` element of the last `day-info` element of this first `share` element as follows:

```
if (doc("shares.xml")/shares/share[1]/@name="INTL")
then do replace value of doc("shares.xml")/shares/share[1]/day-info[1]/high
with doc("shares.xml")/shares/share[1]/day-info[last()]/high
```

Finally the declaration of functions that perform updates is allowed, e.g. the following function accepts an element `$e` as a parameter, and deletes those of its `high` child elements that have attributes with names `day` and `month`:

```
declare updating function remove-high($e as element())
   {
       if ($e/@day and $e/@month) then
        do delete $e/high
   }
```

This function can be called with a parameter that is an XQuery expression that evaluates to a single element,
e.g. `remove-high(doc("shares.xml")/shares/share[1]/day-info[last()])` would delete the `high` node of the last `day-info` node of the first `share`.

### 2.4.2 Update Languages for RDF

There is as yet no standardised language proposal for an RDF update language and we review here the proposals made to date.

RUL [MSCK05] is an RDF update language language based on RQL [KAC$^+$02] and like RQL developed as part of the RDF Suite [RDFSuite]. The execution of an RUL update guarantees that the resulting RDF graph does not violate the semantics of the RDF model nor the semantics of the given RDFS schema. Three update operations are supported, `INSERT`, `DELETE` and `MODIFY`, for both class and property instances. For example, the RUL expression
`INSERT LO(&http://www.dcs.bbk.ac.uk/LOs/MGZN123)`
inserts a new resource with the specified URI as instance of the `LO` class (as given in the RDF Schema in Figure 2.5). As another example, the RUL expression:

```
DELETE LO(X)
FROM {X}dc:type{Y}
WHERE Y = "Book"
USING NAMESPACE dc = &http://purl.org/dc/elements/1.1/
```

deletes all the resources that are instances of the `LO` class and whose `dc:type` property has value `Book`.

MEL (Modification Exchange Language) proposed in [NSST02] was developed as part of the Edutella project [NWQ$^+$02]. MEL provides the expected primitive modification operations, insert, delete and update. Each MEL expression consists of a statement and an optional query constraint. For example, the MEL statement below inserts a new RDF triple by specifying its subject, predicate and object:

```
<edu:Insert rdf:about="#insert_cmd1">
  <edu:newStatement rdf:resource="#insert1_stmt"/>
</edu:Insert>
```

```
<edu:InsertedStatement rdf:about="#insert1_stmt">
  <rdf:subject rdf:resource="http://www.dcs.bbk.ac.uk/LOs/MGZ423"/>
  <rdf:predicate rdf:resource="&dc:title"/>
  <rdf:object rdf:resource="National Geographic Traveller"/>
</edu:InsertedStatement>
```

The `edu:Insert` element defines a new insert command named `insert_cmd1` that defines an RDF triple to be inserted, represented by an RDF resource named `insert1_stmt`. Within `insert1_stmt`, we specify the *subject*, the *predicate* and the *object* of the new triple. MEL is an RDF Schema agnostic language, meaning that that it does not take RDF Schema information into account during the update.

We too have defined an update language for RDF as part of our own ECA language for RDF, discussed in Chapter 5 of this thesis. Our language can INSERT or DELETE an RDF resource specified by its URI, and can INSERT, DELETE or UPDATE a property of a resource. For example,

```
INSERT resource(http://www.dcs.bbk.ac.uk/LOs/BK123) AS INSTANCE OF LO
```

will insert a new resource identified by the specified URI as an instance of the `LO` RDF class;

```
DELETE (http://www.dcs.bbk.ac.uk/LOs/BK187,dc:type,"Book")
```

will delete the property `dc:type` of the specified resource and its `"Book"` literal value; and

```
UPDATE (http://www.dcs.bbk.ac.uk/LOs/BK187,dc:type,"Book" -> "Journal")
```

will update the object of the property `dc:type` of the specified resource from the `"Book"` literal to the `"Journal"` literal.

## 2.5 Event-Condition-Action Rule Languages

Event-Condition-Action rule languages support the definition of rules that automatically perform stated actions in response to the occurrence of stated events provided that stated conditions hold. These rules are called *Event-Condition-Action (ECA)* rules or *triggers* and have three parts: the event, the condition and the action, specified by the general syntax `ON event IF condition DO actions` The *event part* of an ECA rule describes the events that the rule should respond to. When an event specified by the event part of a rule occurs then the rule is *triggered*. If the condition described in its *condition part* holds then the rule *fires*. The *action part* describes the actions to be performed by the rule if the specified event has occurred and the condition has evaluated to true. Execution of these actions may in turn cause further events to occur, which may in turn cause more ECA rules to fire. We refer the reader to [Pat99] and [WC96] for comprehensive discussions of the syntax and semantics of ECA rules and we give in Sections 2.5.1 and 2.5.2 brief summaries, to the level of detail necessary for this thesis.

### 2.5.1 Syntax

Events may be *primitive* or *composite*. Primitive events are atomic events and may be caused by (a) a *structure operation* (e.g. insert a tuple into a table), (b) a *behaviour invocation* where the event is raised by the execution of some user-defined operation (e.g. a program method), (c) *transaction* where the event is raised by some transaction definition command (e.g. abort, commit), (d) *system exception* where the event is raised by a system or application exception, (e) *clock* where the event is raised at some point in time or (f) *external changes* where an incident outside the database raises an event [DBM88].

Composite events consist of a combination of primitive or other composite events as specified by an *event algebra* expression [CM94, GJS92]. Common operators in event algebras include: (i) *disjunction*: event $e_1 \vee e_2$ occurs if either event $e_1$ or event $e_2$ occurs; (ii) *sequence*: event $e_1; e_2$ occurs if $e_2$ occurs having been preceded by $e_1$; and (iii) *conjunction*: event $e_1 \wedge e_2$ occurs when both $e_1$ and $e_2$ have occurred in any order. Most implementations of ECA systems, however, do not support an event algebra as rich as this and they are able to detect an appropriate set of primitive events, with no support for event operators. This may limit the capabilities of reacting to a range of complex situations but it makes the rule execution more easily implemented, optimised and analysed. In this thesis we consider only primitive events on semi-structured data and we do not consider composite events further. One reason for this is because the SeLeNe application, which was the initial motivation of our work, did not require specification and detection of composite events. For the future, we expect that previous work on composite event detection (such as [CM94, GJS92]) would be transferrable to ECA languages and rule processing systems for semi-structured data, and this is an area of future work.

Event occurrences can have associated with them parameters which provide information about the event occurrence, such as the time at which the event occurred, which transaction caused it to occur, and the changes, if any, the event made to the database. These parameters are known as *deltas* and may be referenced by the condition and action parts of the ECA rule.

In the context of database systems, ECA systems may support either *syntactic* or *semantic* triggering. Syntactic triggering means that instances of the event specified in a rule's event part are regarded as having occurred even if they have not modified the database. Semantic triggering means that instances of the event specified in a rule's event part are regarded as having occurred only if they have made changes to the database.

The condition part of an ECA rule determines if the database is in a particular state. It is a query over the database and its environment and has the same semantics as that of the query language used, e.g. SQL or XQuery. The condition may also refer to the state before the execution of the event and the state created after the execution, by making use of the deltas. The condition part can be a single conditional expression expressed in some query language or it can consist of one or more conditional expressions forming a boolean expression with conjunctions, disjunctions and negations.

The action part of a rule describes the logic to be performed if the condition evaluates to true. A rule's actions may change a database state via update operations, invoke an external procedure, generate user notifications or abort a transaction.

ECA rules can facilitate the implementation of various types of reactive functionality. In database systems, they can be used to support integrity constraint specification and enforcement, materialised views, different transaction models, coordination of distributed computations, and version management. ECA rules can also be used to move within a database system functionality that would otherwise have to be provided by the application, for example monitoring and reading stock price changes, propagating load calculations in an architectural design, or recording students' assessment progression in an e-Learning system. ECA rules can also be used in conjunction with external monitoring devices in order to record and respond to events occurring externally to the database. For example, in medical applications they can be used to monitor a patient's condition and react to changes, or in air-traffic control applications to detect potentially dangerous situations and inform the controller.

### 2.5.2 ECA Rule Execution Model

The *execution model* of an ECA rule system specifies how the rules are used by the system at run time. In particular it specifies:

- when the various components of the rule are executed with respect to one another
- what happens when multiple rules are triggered simultaneously

The first aspect is handled by the use of *coupling modes*. A coupling mode specifies the timing of one part of an ECA rule with respect to another. Possible coupling modes for the condition part with respect to the event part are:

- Immediate: The condition is evaluated immediately after the event is detected as having occurred within the current transaction.
- Deferred: The condition is evaluated within the same transaction, but after the last operation in the transaction and just before the transaction commits.
- Decoupled or Detached: The condition is evaluated within a separate transaction.

Possible coupling modes for the actions part with respect to the condition part are similar:

- Immediate: The action is executed immediately after the condition has been evaluated (if the condition is found to be True).
- Deferred: The action is performed within the same transaction, but after the last operation in the transaction and just before the transaction commits.

- Decoupled or Detached: The action is performed within a separate transaction.

Different types of coupling modes may be more or less suitable for certain categories of rules. For example, decoupled execution can help response time since the length of transactions does not grow too large due to rule execution and hence potentially more concurrency is available. Decoupled execution can be useful is situations where the parent transaction aborts, but rule execution is still desired in the child transaction, e.g. updating an access log regardless of whether or not access is granted. Maintaining views can be done immediately for freshness, and either Deferred or Immediate coupling can be used for checking integrity constraints (Immediate for constraints that should never be violated, and Deferred for constraints that need only be satisfied when the database is in a stable state).

The second aspect of the rule execution model is the policy employed for determining which rule to execute next, given that several rules have been previously triggered and are awaiting execution. For rules which were triggered at precisely the same time by the same event occurrence, further information such as priorities can be used: each rule is assigned a unique priority and rules with higher priority are executed earlier that rules with lower priority.

Another important aspect of rule execution is that of *rule granularity*. Usually two types of granularity are supported: *instance-oriented* and *set-oriented* (or *statement-oriented*)[1]. When a set-oriented rule is triggered by some event and is then scheduled for execution, one copy of its action part is placed on the list of pending rules. When an instance-oriented rule is triggered by some event, one copy of its actions part is placed on the pending list for each member of

[1]Sometimes these are also referred to as *instance-level* and *statement-level*. In this thesis we use these terms interchangeably.

affected data set for which the rule's condition evaluates to True. Hence, a single event can give rise to zero, one, or many copies of a triggered rule's actions for instance-oriented rules.

### 2.5.3 ECA Rules on Structured Data

Before the standardisation in SQL3 of a triggering language over relational data [KMC99], numerous research and commercial systems supported the definition of ECA rules over relational data and object-oriented data and we briefly review some of these here.

The POSTGRES Rules Manager [SHP88] is implemented in the POSTGRES relational DBMS [SR86] provided users with the ability to define rules in the DBMS. Two types of rules were supported, depending one whether their event part is a *retrieve* command (a selection) or an *update* command. Two alternative ways to implement the rule system were proposed. The *Tuple Level System* (TLS) performs rule processing on a tuple basis using *rule locks*. Rule locks are special markers, that include the name of the corresponding rule and the type of lock (tuple or relation level), and are placed on the tuples that satisfy the rule condition at the time a rule is defined. When an appropriate event occurs on a tuple that has the appropriate rule locks, then the corresponding rules' actions are executed. The second implementation approach was the *Query Rewrite System* (QRS) used for rules with retrieve events, which rewrites the rules' queries when an appropriate event occurs, generating a new set of queries that retrieve the appropriate results from the database.

The Starburst Rule System [Wid96, Wid92] is the active database extension of the Starburst [Starburst] relational database system developed by IBM. Its rule execution semantics differ from most other active database systems since rule triggering is based on database state transitions rather than tuple or statement-level changes. Each rule is processed at the end of each transaction causing a transition rather than in response to a single data manipulation operation. Rules may also be processed within a transaction if some predefined user commands are issued.

The Ariel system [Han96] is a relational DBMS with a rule system based on the OPS5 [BFKM85] production rules system. The Ariel Rule Language is a production rule language that supports semantic triggering and conditions that check for database state transitions. The Ariel Rule System places emphasis on efficient evaluation of rule conditions by using discrimination networks.

Due to the richness of the object-oriented data model, ECA rules for object-oriented databases often contain additional features compared with ECA rules for relational databases. The principal difference is the availability of a richer set of primitive event types, for example events which are triggered on the invocation of methods or on the creation of objects.

Chimera [CGMR95] is an active database language based on the object-oriented data model. It supports a wide set of events referring to object manipulation (query, creation, update and deletion), object migration (generalisation and specilisation) as well as events caused as a result of system exceptions. Queries, updates and invocations of system or external methods can be a part of rule actions. The rule granularity is set-oriented.

HiPAC [DBB$^+$88] is also an active object-oriented database management system. One of its novelties is that rules are treated as objects that are instances of the *rule* class. Relationships between rules can then be captured, using properties such as generalisation and specialisation between rule classes. The event, condition and action parts of a rule are defined by three corresponding methods of the *rule* class.

SAMOS [GD92] also integrates active and object-oriented features into one system. It provides a rule language supporting primitive and composite event definition. Primitive events can be method invocations, time events, data modification operations or transaction events. Composite events are constructed using conjunctions, disjunctions, negations, sequencing of primitive events or reductions. Reductions allow the signalling of the repeated occurrence of an event only once, e.g. signal an event after every $n$ occurrences of it. Detection of composite events is the main focus of SAMOS and is achieved using *Petri-nets* [GD94] that describe how a number of event occurrences lead to the signaling of a composite event that can, in turn, lead to the triggering of a rule.

EXACT [DJ97] is an active database system built over the ADAM object-oriented DBMS [Pat89]. Its distinctive characteristic is that it allows the user to choose the rule execution model that best fits the semantics of the application it will be used for. Rule control information (e.g. prioritisation) is shared among those rules that support the same aspect of the application logic (e.g. integrity constraint enforcement) and thus share the same execution model. Different subsets of ECA rules can have different execution models.

REACH [BDZH95] is built over OpenOODB [WBT92] and provides a rich set of rule coupling modes and events, including method invocation; flow control events such as begin, end, commit and abort of a transactions; and various time events.

RAP (RAP: The ROCK & ROLL Active Programming System) [DPW99] is an active rule system that extends the ROCK & ROLL deductive object-oriented database system [BPF$^+$94, BPF$^+$95]. ROCK is an imperative programming language used to implement programs and user methods while ROLL is a logic language used to query the database. Since ROLL and ROCK are languages that can be used to express the condition and the action part of the rules respectively, the main focus of RAP is the event language. RAP supports a wide range of primitive events such as insertion, deletion and update of an object, user and system method invocations, attribute fetching, and transaction start, commit and abort. RAP also employs an event algebra that enables the support of composite events. The execution model provided by RAP allows alternative execution characteristics to be encoded in a variety of combinations.

Finally, [RPS95] describes how ECA rules can be added to a functional database programming language. To achieve syntactic integration with ECA rules, the PFL functional DBMS [PS91] is extended to support transactions which are sequences of expressions $e_i$ of the form $e_1; ...; e_n$. An ECA rule comprises three components: an events expression, a condition expression and a sequence of action expressions $a_1; ...; a_n$. During rule execution, the evaluation of an events expression yields a list of values $evs$ that correspond to event occurrences; the removal of the elements of $evs$ that do not satisfy the condition gives a new list of values $evs'$ that form, together with the action expressions, a transaction of the form $a_1\ evs'; ...; a_n\ evs'$ that is then scheduled for evaluation. Transactions and ECA rules are not first class objects in PFL, i.e. they cannot be passed as function parameters nor returned as results by them. When processing a transaction $e_1; ...; e_n$ each element $e_i$ is evaluated in turn and the ECA rules fired by each $e_i$ are evaluated only after the termination of $e_i$ thus ensuring the confluence of expression evaluation. PFL's event, condition and action expressions are expressed in the computationally complete PFL language and are therefore amenable to the same optimisation techniques as any other PFL expression. Also, since PFL's active subsystem is specified in PFL, it is also amenable to formal analysis techniques developed for functional languages such as abstract interpretation and partial evaluation [BPC01].

### 2.5.4 The SQL3 Standard

One of the most significant extensions introduced by the SQL3 standard was the provision of active functionality over relational data by means of *triggers* [KMC99]. To avoid ambiguity in terminology, in this thesis we generally term SQL3 triggers *ECA rules* or just *rules*.

Rule event parts in SQL3 may be triggered by update, insert or delete operations on the database and triggering is syntactic (not semantic). Rules are of two kinds: BEFORE and AFTER. The former conceptually execute their condition and action before the triggering event is executed. The latter execute their condition and action after the triggering event is executed, using an Immediate coupling mode between both event and condition, and between condition and action.

Conditions are evaluated on the database state that the action is executed on, and multiply triggered rules are handled using a last-in-first-out list. Each rule is assigned a unique priority. A rule's action part consists of an SQL statement block that is executed when the rule fires. This, in turn, may cause the triggering of further rules.

The general SQL3 ECA rule syntax is as follows:

```
CREATE TRIGGER <trigger name>
(BEFORE | AFTER)
(INSERT | DELETE | UPDATE [OF <col names>]) ON <subject table>
[REFERENCING {OLD old_name | NEW new_name}]
([FOR EACH (ROW | STATEMENT))
[WHEN <condition>]
<actions>
```

Here the *subject table* is the table whose modification causes the rule to trigger.

The *triggering operation* is the table modification operation whose execution will cause the rule to trigger and it may be an INSERT, a DELETE or an UPDATE statement on a *subject table.*

The *condition* is optional and if omitted it defaults to True. If a rule is activated, the rule *actions* are executed in the specified order.

The number of times a rule is executed is determined by the rule's granularity. In SQL3 this may be *row-level* (i.e. instance-level) or *statement-level*, indicated by the keywords `FOR EACH ROW` and `FOR EACH STATEMENT`, respectively. When neither of these is specified the granularity defaults to statement-level.

During rule execution, both the condition and the action part of the rule have access to the current database state and to also the *old* and the *new* state of the subject table. Two *transition tables* are provided for this purpose. One keeps the old values of the affected rows of the subject table, i.e. rows deleted or row values before they were updated, and one keeps the new values of the affected rows, i.e. rows inserted or row values after they were updated. In case of a row-level trigger, two transition pseudo-variables are available, `NEW` and `OLD`, to refer to the transition tables. Variables can be assigned to the `NEW` and `OLD` pseudo-variables within the `REFERENCING` clause and these variables can then be used within the rule condition and actions.

The SQL3 standard allows the definition of multiple rules that are triggered by the same triggering operation on the same subject table and with the same execution time, BEFORE or AFTER, implying the need for a strategy for ordering the execution of such rules. The SQL3 standard does not define such a strategy, leaving the decision to the system developer. Current relational database vendors, such as Oracle, IBM DB2 and Microsoft SQLServer, apply a rule prioritisation system based on rule definition time, where a rule $r_1$ defined at time $t_1$ will be processed before a rule $r_2$ defined at time $t_2$ if $t_1 < t_2$.

In SQL3, BEFORE triggers are not permitted to themselves change the database state. This is because a BEFORE trigger altering the database may result in a chain of updates, which since they are not visible to the invocations of other triggers may result in a state where no decision can be made on what persists in the database when BEFORE triggers and any subsequent update complete.

### 2.5.5 ECA Rules on Semi-structured Data

Event-Condition-Action rule languages for XML can be divided into two categories depending on the syntactic approach to defining the rules: those following an SQL3-like approach and those not. SQL3-like languages have the advantage that most users are already familiar with the syntax of SQL3 triggers. Languages that do not follow the SQL3 syntax approach use a markup language syntax to define the rules. This may be more complex for the user to learn and encode rules with, but provides the advantage of being easier to parse using standard markup language processing technologies (i.e. DOM, SAX) and also more easily embeddable in other documents or programs defined in another markup language. As will be seen in Chapter 3 onwards, in our XML and RDF ECA languages, we follow the SQL3-like syntax approach, not the markup language approach.

In the first category, there are two languages: Active XQuery described in [BBCC02] and our own XML ECA language described in Chapter 3. Active XQuery uses XPath expressions to define events and XQuery for expressing rule conditions. For the action part, a syntax is used similar to the update extensions proposed for XQuery in [BBFV05]. A review of Active XQuery and comparison with our XML ECA language is presented in Chapter 3.

In the second category there are languages such as ARML (Active Rule Markup Language) [CPHK02] which provides an XML-based rule language for rule sharing among different heterogeneous ECA rule processing systems. The event part of a rule defines the action or the combination of actions, in the form of conjunctions and disjunctions, that will trigger the rule. Due to the primary requirement for rule sharing between various systems, both conditions and actions of rules are abstractly defined as XML-RPC service calls [XMLRPC] which, after rule distribution, are matched to system-specific methods.

Also in the second category is GRML [Wag02], a multi-purpose rule markup language for defining integrity constraints and ECA rules. GRML uses an abstract syntax based on RuleML [RuleML], leaving the mapping to a concrete language for each underlying system implementation. GRML aims to provide mappings for various rule syntaxes in order to allow interoperation between heterogeneous rule systems. GRML also provides constructs for specifying parts of the rules that are to be retrieved from remote data sources or from external data sources such as web services.

Finally, in relational DBMS it is possible to decompose XML documents into a set of relational tables, potentially allowing developers to exploit existing relational triggering functionality in order to define fine-grain triggers over XML data. This is, though, not yet supported by any major relational DBMS.

Turning to ECA rules on RDF data, to our knowledge our RDFTL language is the only such language proposed to date. It employs a path expression syntax developed specifically for the RDF data model to locate different parts of the RDF graph within rule event, condition and action parts. Constructs are available for defining namespaces and for binding variables to the results of path expressions. A set of keywords allows the user to also exploit RDF schema information in path expressions. RDFTL is capable of detecting simple events regarding RDF resource insertion and deletion as well as insertion, deletion and update of RDF properties. In a similar fashion, the action part of RDFTL rules can perform a sequence of actions over RDF data, including insertions and deletions of RDF resources and insertions, deletions and updates of RDF properties. A detailed description of the language is given in Chapter 5.

## 2.6 Summary

In the present chapter we have reviewed the dominant semi-structured data models, XML and RDF, along with their supporting schema definition languages, DTD, XMLSchema and RDF Schema. The XPath and XQuery query languages for XML, as well as the various RDF query language proposals, were reviewed next, followed by a number of XML and RDF update language proposals. We gave an overview of the characteristics of Event-Condition-Action rule systems and the SQL3 standard. We concluded with a discussion of ECA languages for semi-structured data, both XML and RDF.

The discussion in this chapter has presented the current state-of-the-art in the technologies and standards involved in the definition of ECA languages and has revealed the lack of standard update languages for both XML and RDF as well as the absence of extensive research or standardisation regarding ECA rule languages for semi-structured data with the exception of Active XQuery.

In the next chapter, we review our particular ECA language for XML data, giving its syntax and execution semantics, followed by a study of its expressiveness.

# Chapter 3

# XTL: an XML ECA Language

## 3.1 Introduction

XML has became the dominant standard for information exchange in distributed Web-based applications. The increasing support of XML storage and retrieval by the major relational database vendors, such as Oracle, Microsoft SQL Server and DB2, combined with the emergence of native XML repositories, such as eXist [eXist], Xindice [Xindice] and dbXML [dbXML], has increased the need for more advanced features relating to XML data management, similar to those provided for relational data. Given the dynamic nature of many of the applications requiring XML data management [BCP00, BEPR06], reactive functionality over XML data is one of the emerging requirements and ECA rules are one of the candidate technologies for this.

The use of an ECA language for providing reactive functionality over XML data provides an opportunity for new application areas involving XML. The support of actions such as XML data updates, function calls and web service invocations can facilitate a wide variety of reactive behaviours. For example, ECA rules can be used to update the XML data in response to occurrences of events, to notify users of events of interest to them, to perform XML repository maintenance tasks, to collect statistics of user activity over the XML data, to monitor configuration and log file changes, to automatically update a Web Service's definition on UDDI [UDDI] servers in response to a change in a Web Service's WSDL specification [CCMW01], to change the behaviour of an XML script defined with Apache Jelly [Jelly], and so on.

The use of XML for data warehousing necessitates mechanisms to automatically perform incremental maintenance of materialised XML views, to validate, clean and filter the input data streams and to maintain audit trails of the data. By analogy to their use in conventional data warehouses, ECA rules can be used to support this kind of functionality, and this is one of the potential application domains of our XML ECA language (described in Section 3.3).

This chapter is organised as follows. Section 3.2 discusses the major issues, choices and tradeoffs involved in designing an ECA language for semistructured data. Section 3.3 reviews a specific ECA language for XML including its syntax and execution semantics. Section 3.4 conducts for the first time a study of this language's expressiveness in order to determine the class of database transformations that it is able to express. Finally, Section 3.5 compares the language with other similar approaches with respect to its syntax and semantics.

## 3.2 Designing Event-Condition-Action Languages for Semistructured Data

When designing any ECA language, a set of requirements needs to be considered. A first step concerns the choice of the sublanguages for the event, condition and action parts of rules. The language designer needs to decide whether the event language will support composite or only simple events. This choice has its tradeoffs: simple events are easier to analyse and implement but they offer a relatively limited set of incidents that a rule can react to; composite events can offer a rich set of incidents for triggering rules but may prove hard to analyse and implement. If only simple events will be supported then it is important that the event sublanguage is able to detect the set of atomic events that the target applications can generate and on which reactive functionality is required. If composite events will be supported then, in addition, the event algebra that will be required in order to specify the necessary composite events needs to be designed. Event algebras include a range of operators to define combinations of events that are of relevance to an application, such as conjunction, disjunction, sequencing and negation operators. Implementing and analysing event algebras that support such operators can be complex and so knowing the type of composite events that need to be detected and careful choice of the operators is of crucial importance.

Concerning the condition part sublanguage, this too needs to support the appropriate constructs for expressing conditions according to the requirements of the target applications. This may imply, for instance, support of boolean operators and relational operators.

Finally, the action sublanguage needs to be able to perform the set of operations that the target applications require and that are detectable by the event sublanguage. The relationship between the event and the action part of ECA rules allows the encoding of complex application logic through rules that activate each other as a result of cascading triggers. There may also be actions that are not required to be detectable by the event sublanguage. Into this category may fall actions that perform calls to API functions or Web Services which do not affect data within the database.

The expressiveness and complexity of the ECA rule language is another criterion of significant importance. Design choices such as the computational power of the action sublanguage and the coupling mode, affect the expressiveness of the ECA language and its complexity [PV97]. Balancing between expressiveness and simplicity of an ECA language is a challenging task for the language designer. A language that syntactically and semantically simple is easier to implement, analyse and encode rules with, but it may lack some expressiveness, making it harder or even impossible to encode more complex reactive functionality.

The application domains where the ECA language and its processing system are to be deployed plays a crucial role in the above design decisions. By making use of knowledge of the application requirements, the designer can focus on those language and system features which have most significance in the application domains. For example, the design of XTL was motivated by XML data warehousing and that of RDFTL was motivated by P2P e-Learning applications, even though we expect both languages to be more generally applicable.

Turning specifically to XML data, the design of an ECA language for XML clearly needs to take into account the XML data model. The semi-structured nature of XML data gives rise to new issues affecting the use of ECA rules:

- *Event Granularity*: In the relational model, the granularity of data manipulation events is straightforward, since insert, delete, or update events occur when a relation is inserted into, deleted from, or updated, respectively. With XML, specifying the granularity of where data has been inserted, deleted or updated within an XML document becomes more complex and path expressions that identify locations within the document now become necessary.

- *Action Granularity*: Again in the relational model, the effect of data manipulation actions is straightforward, since an insert, delete or update action can only affect tuples in a single relation. With XML, actions now manipulate entire sub-documents, and the insertion or deletion of sub-documents can trigger multiple insert or delete events. For example, using the `shares.xml` document in Figure 2.1, an action that deletes all the `prices` elements in the path `shares/share/day-info/prices` will trigger rules with event parts specifying deletions of `shares/share/day-info/prices` elements and rules with event parts specifying deletions of `shares/share/day-info/prices/price` elements. Also, the choice of an appropriate action language for XML is not obvious, since there is as yet no standard for an XML update language.

Compared to rules for relational databases, ECA rules for XML data are more difficult to analyse, due to the richer types of events and actions. However, rules for XML appear less complex to analyse than rules for object-oriented data which allow arbitrary method calls to trigger events.

The choice of the language to express queries over the XML data in rule event and condition parts is influenced by the emerging standard languages for querying XML data, namely, XPath [W3C99a] and XQuery [W3C05b]. Using XPath as the language for rule event and condition parts has the advantage of simplicity combined with enough expressiveness to select any part of the XML tree. The use of XQuery, on the other hand, adds extra features such as iteration over the nodes of a result set, specification of joins and the use of functions that allow more compact and readable query expressions. The lack, so far, of a standard language for updating XML data requires the design of custom languages to fulfill the data modification requirements of the action part of rules. Extensions of XQuery have been adopted in the ECA languages reported in [BBCC02, BPW02a] in order to express updates in the action parts of the rules and it is likely that the emerging XQuery Update standard [CFR05] will drive XML update language choices in the future.

We note that, triggers on XML data are supported by all the major DBMS vendors and also by some native XML repository vendors [eXist, Xindice, dbXML]. However, these are confined to document-level triggering, and only events concerning insertion, deletion or update of an entire XML document can be detected, thus limiting the ability to define rules that trigger in response to data modification within the stored XML documents.

## 3.3 XTL: an Event-Condition-Action language for XML

In the present section we give an overview of the ECA language for XML proposed by Bailey, Poulovassilis, Wood in [BPW02a]. This is the language supported by our XML ECA rule processing system described in Chapter 4. In this thesis, we name this language XTL (XML Triggering Language) although it was not named in [BPW02a].

The design of XTL predated the SeLeNe project and was motivated primarily by the observation that XML can potentially become a useful tool in data warehousing applications. One of the design goals was to keep the language as simple as possible, making it easy to analyse. As a result, fragments of XPath and XQuery were chosen to define the query and update sublanguages for XTL rules. The expressiveness of XTL is examined for the first time in Section 3.4.

### 3.3.1 Syntax

The syntax of XTL rules follows the SQL3 syntactic approach, consisting of an event, condition and action part. The general format of an XTL rule is:

`on` *event* `if` *condition* `do` *actions*

Fragments of XPath and XQuery are used to specify the event, condition and actions parts of rules. XPath is used for selecting and matching fragments of XML documents within the event and condition parts while XQuery is used within insertion actions, where there is a need to be able to construct new XML fragments.

The *event* part of an XTL rule is an expression of the form

`INSERT` $e$

or

`DELETE` $e$

where $e$ is a *simple XPath expression* (see below) which evaluates to a set of nodes. The rule is triggered if this set of nodes includes any node in a new XML fragment, in the case of an insertion, or in a deleted fragment, in the case of a deletion.

The system-defined variable `$delta` is available for use within the condition and actions parts of the rule, and its set of instantiations is the set of new or deleted nodes returned by $e$.

The *condition* part of a rule is either the constant `TRUE`, or one or more simple XPath expressions connected by the boolean connectives `and`, `or`, `not`.

The *actions* part of a rule is a sequence of one or more actions:

$action_1;\ \ldots;\ action_n$

where each $action_i$ is an expression of one of the following three forms:

`INSERT` $r$ `BELOW` $e$ `BEFORE` $q$

`INSERT` $r$ `BELOW` $e$ `AFTER` $q$

DELETE $e$

Here, $r$ is a *simple XQuery expression* (see below), $e$ is a simple XPath expression and $q$ is either the constant `TRUE` or an *XPath qualifier* (see below).

In an `INSERT` action, the expression $e$ specifies the set of nodes, $N$, immediately below which new XML fragment(s) will be inserted. These fragments are specified by the expression $r$. If $e$ or $r$ references the `$delta` variable, then one XML fragment is constructed for each instantiation of `$delta` for which the rule's condition evaluates to True. If neither $e$ nor $r$ references `$delta`, then a single fragment is constructed. The expression $q$ is an XPath qualifier which is evaluated on each child of each node $n \in N$. For insertions of the form `AFTER` $q$, the new fragment(s) are inserted after the last sibling for which $q$ is True, while for insertions of the form `BEFORE` $q$, the new fragment(s) are inserted before the first sibling for which $q$ is True. The order in which new fragments are inserted is non-deterministic.

In a `DELETE` action, the expression $e$ specifies the set of nodes which will be deleted (together with their descendant nodes). Again, $e$ may reference the `$delta` variable.

**Example** Consider an XML repository containing the XML document `shares.xml` given in Figure 2.1 that contains information about share prices on a Stock Exchange.

Suppose that this document is updated in response to external events received from a share price information service. In particular, an insertion event will arrive periodically with the current price for share `INTL`. For example, such an insertion event, $Ev$, might be the following update, which inserts the new share price of `123.75` after the last share price currently recorded:

```
INSERT <price time="09:15">123.75</price>
```

```
BELOW document('shares.xml')/shares/share[@name="INTL"]/
            day-info[@day="03"][@month="03"]/prices
AFTER TRUE
```

The following ECA rule, $r_1$, checks whether the daily high needs to be updated in response to a new price insertion in some share:

```
on INSERT document('shares.xml')/shares/share/day-info/prices/price
if $delta > $delta/../../high
do DELETE $delta/../../high;
   INSERT <high>$delta/text()</high>
          BELOW $delta/../.. AFTER prices
```

Here, the system-defined `$delta` variable is bound to the newly inserted `price` node detected by the event part of the rule. The rule's condition checks that the value of this `price` node is greater than the value of the `high` node under the same `day-info` node. The action then deletes the existing `high` node and inserts a `high` node whose value is that of the newly inserted `price`.

The insertion event $Ev$ above would trigger this rule $r_1$, which would then update the daily high of share `INTL` to `123.75`.

The following ECA rule, $r_2$, similarly checks whether the monthly high price for a share needs to be updated in response to an insertion of a new daily high price:

```
on INSERT document('shares.xml')/shares/share/day-info/high
if $delta > $delta/../../month-info[@month=$delta/../@month]high
do DELETE $delta/../../month-info/high;
   INSERT $delta
     BELOW $delta/../../month-info[@month=$delta/../@month]
     BEFORE TRUE
```

In the `INSERT` action of this rule, a copy of the `high` node whose insertion triggered the rule is inserted as the first child of the corresponding `month-info` node.

The event $Ev$ above would trigger rule $r_1$, and the second action of $r_1$ would in turn trigger rule $r_2$. However, the condition of $r_2$ would then evaluate to False and so its action would not be executed.

The XPath and XQuery expressions appearing in our ECA rules are restrictions of the full XPath and XQuery languages, to what are termed *simple* XPath and XQuery expressions in [BPW02a]. This simplicity combined with reasonable expressiveness makes these fragments easier to analyse than the full XPath and XQuery languages, as discussed in [BPW02a, BPW02b].

The XPath fragment used disallows a number of features of the full XPath language, in particular the use of any axis other than the child, parent, self or descendant-or-self axes and of all functions other than `document()` and `text()`. Thus, the syntax of a *simple XPath expression* $e$ is given by the following grammar, where $s$ denotes a string and $n$ denotes an element or attribute name:

$$\begin{array}{lll}
e & ::= & \text{`}\texttt{document(}\text{'}\ s\ \text{`}\texttt{)}\text{'}\ ((\ \text{`}\texttt{/}\text{'}\ \mid\ \text{`}\texttt{//}\text{'}\ )\ p)\ \mid \\
 & & \text{`}\texttt{\$delta}\text{'}\ (\text{`}\texttt{[}\text{'}\ q\ \text{`}\texttt{]}\text{'})^*\ ((\ \text{`}\texttt{/}\text{'}\ \mid\ \text{`}\texttt{//}\text{'}\ )\ p)? \\
p & ::= & p\ \text{`}\texttt{/}\text{'}\ p\ \mid\ p\ \text{`}\texttt{//}\text{'}\ p\ \mid\ p\ \text{`}\texttt{[}\text{'}\ q\ \text{`}\texttt{]}\text{'}\ \mid\ n\ \mid\ \text{`}\texttt{*}\text{'}\ \mid \\
 & & \text{`}\texttt{@}\text{'}n\ \mid\ \text{`}\texttt{@*}\text{'}\ \mid\ \text{`}\texttt{.}\text{'}\ \mid\ \text{`}\texttt{..}\text{'}\ \mid\ \text{`}\texttt{text()}\text{'} \\
q & ::= & q\ \text{`}\texttt{and}\text{'}\ q\ \mid\ q\ \text{`}\texttt{or}\text{'}\ q\ \mid\ e\ \mid\ p\ \mid\ (p \mid e \mid s)\ o\ (p \mid e \mid s) \\
o & ::= & \text{`}\texttt{=}\text{'}\ \mid\ \text{`}\texttt{!=}\text{'}\ \mid\ \text{`}\texttt{<=}\text{'}\ \mid\ \text{`}\texttt{<}\text{'}\ \mid\ \text{`}\texttt{>=}\text{'}\ \mid\ \text{`}\texttt{>}\text{'}
\end{array}$$

Expressions enclosed in '`[`' and '`]`' in an XPath expression (generated by $q$ in the grammar above) are called *qualifiers*. So a simple XPath expression starts by establishing a context, either by a call to the `document` function followed by a path expression $p$, or by a reference to the variable `$delta` (the only variable allowed) followed by optional qualifiers $q$ and an optional path expression $p$. Note that a qualifier $q$ can comprise a simple XPath expression $e$.

The XQuery fragment adopted disallows the use of full FLWR expressions, essentially permitting only the **return** part of such an expression. The syntax of a *simple XQuery expression* $r$ is given by the following grammar:

$$\begin{array}{rcl} r & ::= & e \mid c \\ c & ::= & \text{`}\texttt{<}\text{'}\ n\ a\ (\text{`}\texttt{/>}\text{'} \mid (\text{`}\texttt{>}\text{'}\ t{*}\ \text{`}\texttt{</}\text{'}\ n\ \text{`}\texttt{>}\text{'})) \\ a & ::= & (n\ \text{`}\texttt{= "}\text{'}\ (s \mid e')\ \text{`}\texttt{"}\text{'}\ a)? \\ t & ::= & s \mid c \mid e' \\ e' & ::= & \text{`}\{\text{'}\ e\ \text{`}\}\text{'} \end{array}$$

Thus, an XQuery expression $r$ is either a simple XPath expression $e$ (as defined above) or an element constructor $c$. An element constructor is either an empty element or an element with a sequence of element contents $t$. In each case, the element can have a list of attributes $a$. An attribute list $a$ can be empty or is a name equated to an attribute value followed by an attribute list. An attribute value is either a string $s$ or an *enclosed expression* $e'$. Element contents $t$ is one of a string, an element constructor or an enclosed expression. An enclosed expression $e'$ is a simple XPath expression $e$ enclosed in braces. The braces indicate that $e$ should be evaluated and the result inserted at the position of $e$ in the element constructor or attribute value.

### 3.3.2 Rule Execution Model

We now describe the rule execution model of XTL, as defined in [BPPW04]. The input to the execution is a *schedule* `s` and an *XML repository* `db`. The schedule consists of a list of pairs $(action_{i,j}, delta_i)$, where $action_{i,j}$ is the $j^{th}$ action within the actions part of rule $r_i$ and $delta_i$ is a set of instantiations for the `$delta` variable of rule $r_i$ for which $r_i$'s condition evaluated to True.

Rules whose event parts reference the same XML document can potentially be triggered by the same update event on that document. To disambiguate the effect of such rules, all rules whose event parts are insertions on the same document need to be totally ordered, as do all rules whose event parts are deletions on the same document. The relative priorities of such rules are specified by the user when defining a new rule. This is almost a global priority model, as all rules whose event parts refer to insertions on the same document are totally ordered, as are all rules whose event parts refer to deletions on the same document. This priority model is simpler to specify, implement and enforce than a finer-granularity ordering of sets of rules would have been.

The schedule which initiates rule execution consists of an action and a set of instantiations for the `$delta` variable upon which this action is to be applied, i.e. the initial schedule is a singleton list of the form `[(action,delta)]`. The following pseudocode expresses how this update request is handled:

```
while s != [] do {
   (a,delta)    := head (s);
   s            := tail (s);
   (changes,db) := updateDB (db,a,delta);
   for each rule r_i in order of increasing priority do {
      if changes[i] != {} then {
         (value,delta) := evalCondition(i,changes[i],db);
         if value = True then
         for j := noOfActions[i] downto 1 do
            s := (action[i,j],delta):s
      }
   }
}
```

In the above pseudocode, the function `head` returns the first element of a list and the function `tail` returns a list minus its first element.

The function `updateDB` executes the action `a` that was at the head of the schedule. If `a` does not reference the `$delta` variable, this update is performed just once on the repository `db`. If `a` does reference the `$delta` variable, a set of updates is generated by substituting occurrences of `$delta` within `a` by each member of `delta`. Thus if $n$ is the cardinality of `delta`, $n$ updates will be generated[1]. These updates are then performed in an arbitrary order on the repository.

`updateDB` returns a pair `(changes,db)`, where `db` is the new repository resulting from the update and `changes` is an array such that `changes[i]` is the set of newly inserted or newly deleted nodes corresponding to the event part of rule `r_i`. In particular, if `a` is an insertion then for each `r_i` which may be triggered by `a`, the event part of `r_i` is evaluated on the repository after `a` is executed, and `changes[i]` is the intersection of this result and the new nodes inserted by `a`. If `a` is a deletion then for each `r_i` which may be triggered by `a`, the event part of `r_i` is evaluated on the repository before `a` is executed, and `changes[i]` is the intersection of this result and the nodes that are subsequently deleted by `a`.

The function `evalCondition` evaluates rule `r_i`'s condition and there are two possible cases:

(i) If the `$delta` variable occurs in the condition, then the condition is evaluated once for each member of `changes[i]`, and the subset of `changes[i]` for which it evaluates to True is determined. The variable `delta` is set to this subset. If `delta` is non-empty, then the variable `value` is set to `True`; otherwise it is set to `False`.

(ii) If the `$delta` variable does not occur in the condition, then the condition is

[1]$n$ is guaranteed to be finite due to the syntax of the update language, which does not allow infinite new XML fragments to be created, and the fact that there are a finite number of ECA rules.

evaluated just once and the variable `value` is set to the result. The variable `delta` is set to `changes[i]`.

`noOfActions[i]` is the number of actions in the actions part of rule `r_i`, and `actions[i,j]` is the $\mathtt{j}^{th}$ action of rule `r_i`. The loop `for j:=noOfActions[i] downto 1 do ...` ensures that the actions of a given rule are placed in the right order onto the schedule. Each such action `action[i,j]` is paired with that rule's `delta` and prefixed to the current schedule, by the statement `s := (action[i,j],delta):s`.

Rules are considered in increasing order of their priority in the outer `for` loop. Thus, the actions of higher-priority rules that have fired will be placed onto the schedule in front of the actions of lower-priority rules.

The execution proceeds in this manner until the schedule becomes empty. Non-termination of rule execution is a possibility and thus development of static rule analysis techniques is important to aid the design of 'well-behaved' rules. Such techniques are discussed in [BPPW04] and we do not consider them further in this thesis.

There are a number of observations made in [BPPW04] regarding the above rule execution model:

- Triggering is semantic, not syntactic, since a rule `r_i` is triggered only if `changes[i]` is not empty[2].

- The event/condition coupling mode and the condition/action coupling modes are both `Immediate`, since conditions are evaluated immediately after an event becomes true, and the actions of rules that have fired as a result of the current set of updates are placed at the head of the schedule.

[2]We recall from Section 2.5.1 that syntactic triggering implies that instances of the event specified in a rule's event part are regarded as having occurred even if they have not modified the database. Semantic triggering means that instances of the event specified in a rule's event part are regarded as having occurred only if they have made changes to the database.

- Rule conditions are evaluated against the repository state in which the rule was triggered, unlike in SQL3 where conditions are evaluated against the database state that the action(s) will be executed on.

  It is possible to simulate the behaviour of SQL3 using XTL rules by adding the conditions as additional qualifiers to the XPath expression $e$ that is part of `INSERT` and `DELETE` actions, and setting the condition part of the rule to `TRUE`. For example the rule $r_1$ given in Section 3.3.1 above, would be rewritten as follows to simulate the behaviour of SQL3:

  ```
  on INSERT document('s.xml')/shares/share/day-info/prices/price
  if TRUE
  do DELETE $delta[. > ./../../high]/../../high;
     INSERT <high>$delta/text()</high>
            BELOW $delta[. > ./../../high]/../.. AFTER prices
  ```

- Both *document-level* and *instance-level* triggering are supported in XTL, depending on the occurrence of `$delta` in the condition and action parts of a rule, by analogy to statement and row-level triggers in SQL3:

  – If there is no occurrence of `$delta` in the condition or the action, the action is executed once if the condition is True — this is document-level triggering.

  – If `$delta` occurs in the action (and possibly in the condition), the action is executed once for each possible instantiation of `$delta` for which the condition is True — this is instance-level triggering.

  – `$delta` is an instance-level variable, as implied by the definition of the fragment of XPath that XTL employs (see Section 3.3.1), and it cannot denote a collection of instances. This choice was made for XTL path expressions in the interests of keeping the language as simple as possible, in order to facilitate rule analysis while still retaining a reasonable level of expressiveness.

## 3.4 Expressiveness of XTL

Due to the lack of any theoretical analysis to date of the expressiveness of XML query and update languages, in this section we exploit the ability of XTL to express transformations over *relational* data as an indication of its expressiveness. For this, we explore theoretical studies regarding the expressiveness of database languages over relational data. We give below a relational to XML data conversion algorithm which provides the necessary mapping between relational and XML data model constructs in order to make the appropriate transformations of the expressiveness criteria from relational data to XML data.

Given this mapping between relational and XML data, we aim to show that XTL is at least as powerful as the $while_{new}$ language, which is known to be query complete over relational data [AVH96]. To show this we emulate the constructs of $while_{new}$ using XTL rules.

We first review expressiveness of languages over the relational data model. These results are then used in order to draw conclusions regarding the expressiveness of XTL. Most of the information presented in Section 3.4.1 has been drawn from [AVH96].

### 3.4.1 Review of Relational Language Expressiveness

A *query* is a mapping from instances of a fixed input schema to instances of a fixed answer schema that is computable and generic, meaning that it is "implementable" by a Turing Machine and that it depends only on information provided by the input instance, respectively [AVH96]. The *while* query language incorporates *while* statements as well as destructive assignment statements of the form $R := E$, where $E$ is relational algebra expression and $R$ a relational variable of the same type as the result of $E$. The semantics of such an assignment statement is that the value of $R$ becomes the result of evaluating the expression $E$ on the current database state. *while* statements have the form:

```
while change do
   begin
    (loop body)
   end
```

A *while* statement repeats as long as execution of the body causes some change to some relation. The body can contain a sequence of assignment statements and other nested *while* statements.

The *while* language is bounded by polynomial space, and it is complete in PSPACE, although there are queries in PSPACE, such as *even* (which determines if a relation has an even number of tuples), that it cannot express.

*while* cannot go beyond the PSPACE because (1) throughout the computation it uses only values from the input, and (2) it uses relations with fixed arity. One of the ways to go beyond this space complexity barrier is by relaxing one of (1) or (2). We refer the reader to [AVH96] for a language that relaxes (2). Relaxing (1) allows the creation of new values that are not present in the input. An extension of *while* that allows the creation of new values during the computation is the $while_{new}$ language. The modifications of *while* to obtain $while_{new}$ are as follows:

(i) There is a new instruction $R := new(S)$, where $R$ and $S$ are relations and $arity(R) = arity(S) + 1$;

(ii) There is a looping construct of the form *while $R$ do $s$*, with $R$ being a relational variable and $s$ being a sequence of assignment statements or loops. The loop terminates when $R$ is empty.

The *new* instruction extends each tuple of $S$ with a new distinct value from the value domain of the database. The new value must not occur in the input, the current database or the program itself. So, assuming that we have a relation scheme $S[a_1, a_2, ...a_n]$, where $a_i,\ i \in [1, n]$ are its attributes, a relation with scheme $R[a_1, a_2, ...a_n, a_{n+1}]$, where $a_{n+1}$ is a new attribute name not appearing in $a_1, ..., a_n$ set, is obtained by extending each tuple of $S$ by a distinct new value.

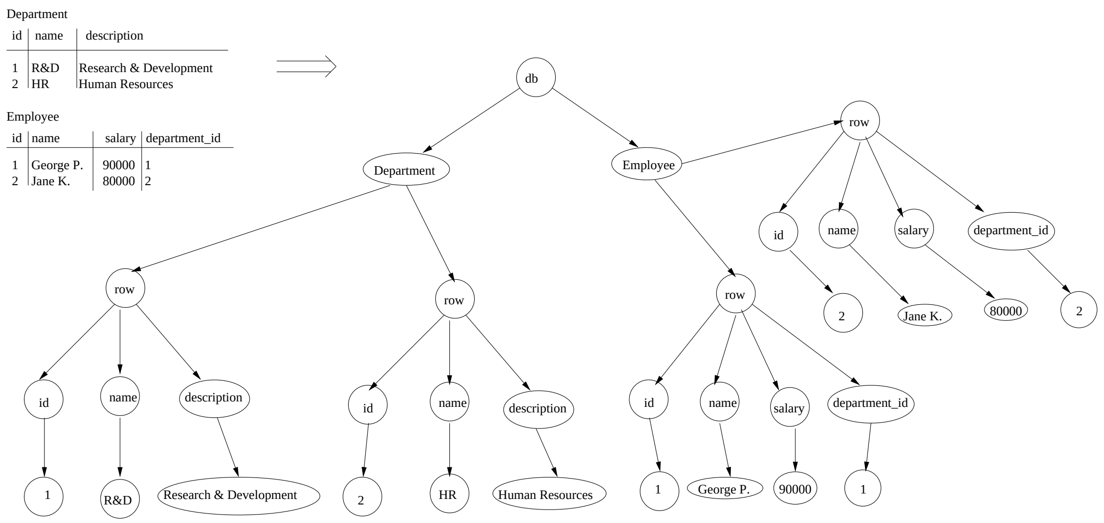
$while_{new}$ programs may give several possible outcomes depending on the choice of new values, something that does not conform with the requirement that a query should be deterministic. Therefore only *well-behaved* $while_{new}$ programs, i.e. those whose answers do not contain newly created values, are considered. Note that although well-behaved $while_{new}$ programs are deterministic as far as their final answer is concerned, they are not deterministic as far as their intermediate results are concerned since they may contain new values. As shown in [AVH96], $while_{new}$ can express all queries over relational data.

### 3.4.2 Relational to XML Conversion

Let us assume a relational database schema $\mathbf{R}$ containing a set of relation schemes $R_1, R_2, ...R_n$ with $R_i$ having attributes $attr_{i1}...attr_{ij_i}$, and $r_i$ being a relation over $R_i$, for $1 \leq i \leq n$.

Given a relational database $D = \{r_1, r_2, ...r_n\}$, an algorithm, similar to the one presented in [ABS00], converts $D$ to an XML tree $d$ starts by creating the root element named *db*. For each relation $r_i$ in $D$ it creates an XML element node with the same name as $R_i$ as a child of the `root` node. For each tuple $t$ in $r_i$ it creates an XML element node named `row` as a child of the node corresponding to $R_i$. For each attribute $attr_{ik}$ of $R_i$ a new element named $attr_{ik}$ is added as a child of each `row` element. For each $1 \leq k \leq j_i$, the value of $attr_{ik}$ in tuple $t$ is represented by a text child node of the node corresponding to $attr_{ik}$.

Figure 3.1 gives an example that illustrates this relational to XML data conversion.



Department

| id | name | description |
|---|---|---|
| 1 | R&D | Research & Development |
| 2 | HR | Human Resources |

Employee

| id | name | salary | department_id |
|---|---|---|---|
| 1 | George P. | 90000 | 1 |
| 2 | Jane K. | 80000 | 2 |

Figure 3.1: Example data conversion

### 3.4.3 Expressiveness of our Language

As stated in Section 3.4.1 above, the *while* language incorporates two constructs: a destructive assignment $R := E$, where $R$ is a relation and $E$ a relational algebra expression and a `while` loop. As part of the `while` loop body it allows a sequence of assignments or `while` statements.

To emulate assignment statements, we have to show that using XTL rules we can express the relational algebra operations: (a) union ($R_1 \cup R_2$), (b) difference ($R_1 - R_2$), (c) project ($\pi_{a_1,\ldots,a_N} R_1$), (d) select ($\sigma_C R_1$) and (e) cross product ($R_1 \times$

$R_2$). The conversion algorithm presented in Section 3.4.2 gives the correspondence between relational and XML data.

For the purposes of the emulation we assume that there exists a temporary XML file named `temp.xml` into which we perform insertions and deletions of XML nodes that act as flags. These flags are used to emulate conditionals, as triggering events for our rules, and to control sequences of statements. It is assumed that the flag names conform to a reserved naming scheme depending on the language constructs that are emulated. So, for example, when emulating the union operation the flag names are of the form `union_flag_i` where `i` is an integer. When a new flag needs to be used, the integer `i` is given the smallest unused value. We also assume that the XML data is stored in the file `db.xml`.

The set of rules that follow show how each of the five basic relational algebra operations can be emulated using XTL[3]:

- Union ($R := R_1 \cup R_2$): Given two relations $R_1$ and $R_2$ represented in XML as shown Figure 3.1, the following rules perform a destructive assignment to $R$ of the result of the union of $R_1$ and $R_2$. In the rules below, the first two rules check if the element `R` (corresponding to relation $R$) exists or not. If it does, its contents are deleted, to emulate the destructive assignment; if it does not exist, it is created. The contents of both $R_1$ and $R_2$ are then copied to $R$ by the third rule, checking that no data duplication occurs.

```
r1:
ON INSERT document('temp.xml')/flags/union_flag_i
IF not document('db.xml')/db/R
DO INSERT <R/> BELOW document('db.xml')/db AFTER TRUE;
   INSERT <union_flag_i+1/>
```

[3]For reference purposes, we prefix each rule with `ri` to indicate the `i`th rule in the overall rule set

```
     BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('temp.xml')/flags/union_flag_i;

r2:
ON INSERT document('temp.xml')/flags/union_flag_i
IF document('db.xml')/db/R
DO INSERT <union_flag_i+1/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('db.xml')/db/R/*;
   DELETE document('temp.xml')/flags/union_flag_i;

r3:
ON INSERT document('temp.xml')/flags/union_flag_i+1
IF TRUE
DO INSERT document('db.xml')/db/R1/row
     BELOW document('db.xml')/db/R AFTER TRUE;
   INSERT
     document('db.xml')/db/R2/row
              [not (.=document('db.xml')/db/R1/row)]
     BELOW document('db.xml')/db/R AFTER TRUE;
   DELETE document('temp.xml')/flags/union_flag_i+1;
```

Here, '=' is a value-based equality operator. Thus, the expression `.=document('db.xml')/db/R1/row` in the second insert action of `r3`, compares the string that results from the concatenation of the text nodes, in document order, in the result set of the left expression to the string that results from the concatenation of the text nodes in the right expression, following the same concatenation order.

- Project ($R := \pi_{<a_1,\ldots,a_N>} R_1$): Similarly to union, two rules ensure that $R$ exists and is empty. These rules are exactly the same as above, except the name of the flag is now `project_flag_i`, and thus they are omitted below. The contents of `R1` are deleted by rule `r1` below, before the selected attributes are projected to `R` and the contents of `R1` are restored by rule `r2`.

```
r1:
ON INSERT document('temp.xml')/flags/project_flag_i+1
IF TRUE
DO INSERT <project_flag_i+2/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('db.xml')/db/R1/row;
   DELETE document('temp.xml')/flags/project_flag_i+1;

r2:
ON DELETE document('db.xml')/db/R1/row
IF document('temp.xml')/flags/project_flag_i+2
DO INSERT <row>
            $delta/a1
            ...
            $delta/aN
          </row>
    BELOW document('db.xml')/db/R[not (./row = $delta)]
    AFTER TRUE ;
   INSERT $delta BELOW document('db.xml')/db/R1 AFTER TRUE ;
   DELETE document('temp.xml')/flags/project_flag_i+2;
```

- Select ($R := \sigma_C R_1$): Two rules again ensure that `R` exists and is empty. These are the same as for union except that the name of the flag is now `select_flag_i`, and they are omitted. The condition $C$ is a boolean expression with conjunctions, disjunctions and negations of comparison expressions containing the operators $=, \neq, >=, >, <=, <$. These are covered by our language syntax described in Section 3.3.1. The rule below applies the condition to the contents of `R1` and inserts the results under `R`.

```
r1:
ON INSERT document('temp.xml')/flags/select_flag_i+1
IF TRUE
DO INSERT <select_flag_i+2/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   INSERT document('db.xml')/db/R1/row[C]
     BELOW document('db.xml')/db/R AFTER TRUE ;
   DELETE document('temp.xml')/flags/select_flag_i+1;
```

- Difference ($R := R_1 - R_2$): Two rules again ensure that `R` exists and is empty. Rule `r1`, below, copies the contents of `R1` to the empty `R`. Rule `r2` then checks each of the rows just inserted into `R` and, if found equal to any row of `R2`, the row is deleted from `R`.

```
r1:
ON INSERT document('temp.xml')/flags/difference_flag_i+1
IF TRUE
DO INSERT <difference_flag_i+2/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   INSERT document('db.xml')/db/R1/row
```

```
      BELOW document('db.xml')/db/R AFTER TRUE ;
   DELETE document('temp.xml')/flags/difference_flag_i+1;


r2:
ON INSERT document('db.xml')/db/R/row
IF ($delta = document('db.xml')/db/R2/row) and
   (document('temp.xml')/flags/difference_flag_i+2)
DO DELETE $delta;
   DELETE document('temp.xml')/flags/difference_flag_i+2;;
```

- Product ($R := R_1 \times R_2$): Again, two rules ensure that R exists and is empty. To emulate the cross product of R1 and R2 we use the following sequence of rules r1 to r7. In order to illustrate the results of each processing step, we use as an example the trees R1 and R2 shown in Figure 3.2. Rule r1 deletes the contents of R1 and R2 if both are non-empty:

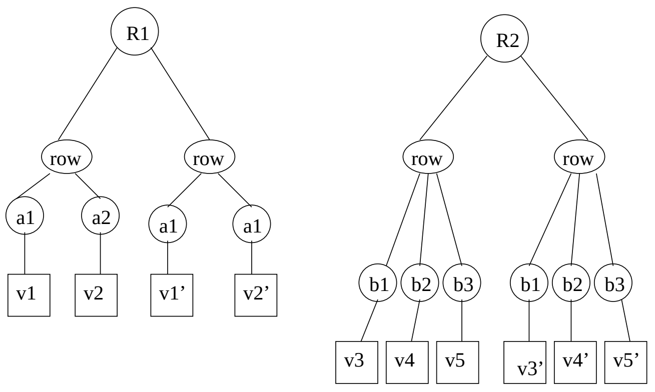


Figure 3.2: Relations R1 and R2

```
r1:
ON INSERT document('temp.xml')/flags/product_flag_i+1
```

```
IF (document('db.xml')/db/R1/row) and
   (document('db.xml')/db/R2/row)
DO INSERT <product_flag_i+2/>
        BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('db.xml')/db/R1/row ;
   DELETE document('db.xml')/db/R2/row ;
   DELETE document('temp.xml')/flags/product_flag_i+1;
```

Rule `r2` copies the deleted rows from `R1` under `R` and also back under `R1`. Rule `r3` does the same for the deleted rows of `R2` but copies them now under every `R/row` element as well as under `R2`:

```
r2:
ON DELETE document('db.xml')/db/R1/row
IF (document('temp.xml')/flags/product_flag_i+2)
DO INSERT <product_flag_i+3/>
        BELOW document('temp.xml')/flags AFTER TRUE;
   INSERT $delta BELOW document('db.xml')/db/R AFTER TRUE;
   INSERT $delta BELOW document('db.xml')/db/R1 AFTER TRUE;
   DELETE document('temp.xml')/flags/product_flag_i+2;


r3:
ON DELETE document('db.xml')/db/R2/row
IF (document('temp.xml')/flags/product_flag_i+3)
DO INSERT <product_flag_i+4/>
        BELOW document('temp.xml')/flags AFTER TRUE;
   INSERT $delta BELOW document('db.xml')/db/R/row AFTER TRUE;
   INSERT $delta BELOW document('db.xml')/db/R2 AFTER TRUE;
   DELETE document('temp.xml')/flags/product_flag_i+3;
```

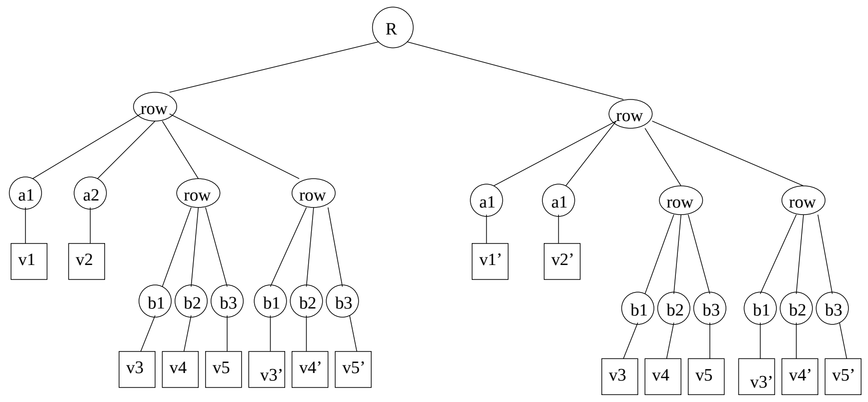


Figure 3.3: R after the execution of `r3`

After the execution of `r2` and `r3`, `R` is as shown in Figure 3.3. The insertion of data by `r3` under `R/row` triggers `r4` which deletes all the elements under `R/row`, an action that triggers `r5`. `r5` puts every element that was deleted by `r4` under each `row` below the `R/row` from which the current element was deleted.

```
r4:
ON INSERT document('db.xml')/db/R/row/*
IF document('temp.xml')/flags/product_flag_i+4
DO INSERT <product_flag_i+5/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('db.xml')/db/R/row/*;
   DELETE document('temp.xml')/flags/product_flag_i+4;

r5:
ON DELETE document('db.xml')/db/R/row/*
```

```
IF document('temp.xml')/flags/product_flag_i+5
DO INSERT <product_flag_i+6/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   INSERT $delta
     BELOW document('db.xml')/db/R/row/row[..=$delta/..]
     AFTER TRUE;
   DELETE document('temp.xml')/flags/product_flag_i+5;
```

After the execution of `r4` and `r5`, `R` is as shown in Figure 3.4. `r6` now deletes the third level of nested `row` elements (see Figure 3.5), an action that triggers `r7`. `r7` creates an auxiliary element node `R'` to temporarily store the second level of nested `row` contents of `R`, before they are placed back into `R`, after the deletion of the first level `row` element nodes of `R`:

```
r6:
ON INSERT document('db.xml')/db/R/row/row/*
IF document('temp.xml')/flags/product_flag_i+6
DO INSERT <product_flag_i+7/>
     BELOW document('temp.xml')/flags AFTER TRUE;
   DELETE document('db.xml')/db/R/row/row/row;
   DELETE document('temp.xml')/flags/product_flag_i+6;

r7:
ON DELETE document('db.xml')/db/R/row/row/row
IF document('temp.xml')/flags/product_flag_i+7
DO INSERT <R'/> BELOW document('db.xml')/db;
   INSERT document('db.xml')/db/R/row/row
     BELOW document('db.xml')/db/R' AFTER TRUE;
```

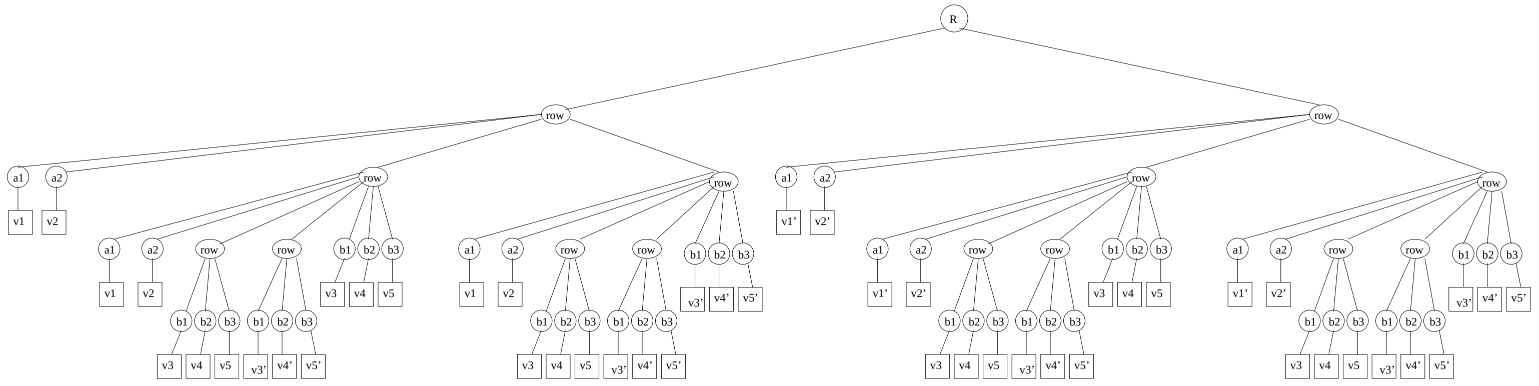


Figure 3.4: R after the execution of r5

```
DELETE document('db.xml')/db/R/row ;
INSERT document('db.xml')/db/R'/row
  BELOW document('db.xml')/db/R AFTER TRUE ;
DELETE document('db.xml')/db/R';
DELETE document('temp.xml')/flags/product_flag_i+7;
```

Figure 3.6 shows the final form `R` after the execution of `r7`.

In conclusion, the above set of rules show that XTL can support destructive assignment statements of the form $R := E$, where $E$ is a relational algebra expression.

To emulate a statement sequence $s_1; s_2$ we create a "link" rule that is achieved in response to the deletion of the flag in the last action of the last rule emulating the statement $s_1$. Upon activation, this link rule inserts the appropriate flag to trigger the first rule emulating statement $s_2$. For example for the sequence $R := R_1 \cup R_2; R' := R - R1$, a link rule would be created that is triggered in response to the deletion of the `union_flag_i+1` from the first statement, and upon activation inserts a flag `difference_flag_j` in order to trigger the rule that emulates the second statement:

```
ON DELETE document('temp.xml')/flags/union_flag_i+1
IF TRUE
DO INSERT document('temp.xml')/flags/difference_flag_j
```

For emulating a *while* loop over a relational variable, three ECA rules can be used, `r1`, `r2` and `r3` below. The first two rules are both triggered in response to the insertion of an XML element `flag_while_i` into the auxiliary `temp.xml` file, but their conditions express two opposite situations; one evaluates to true if the relation $R$ has at least one row, while the other holds when $R$ has no rows.

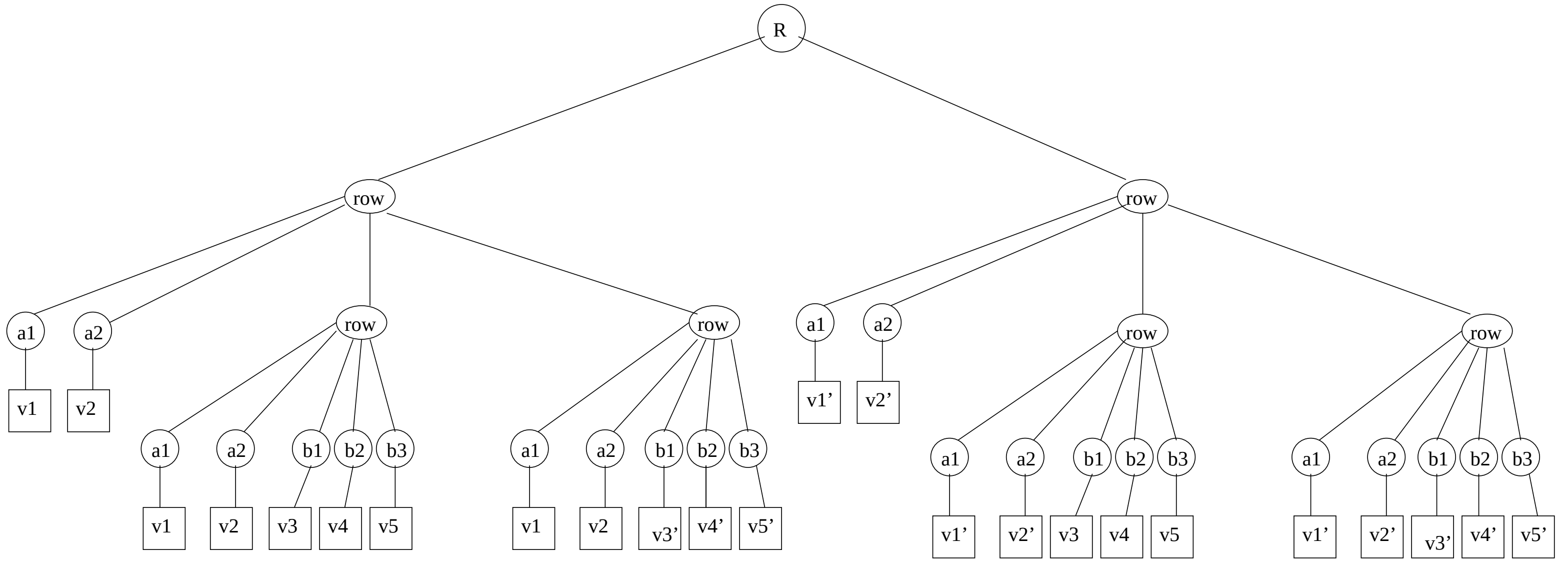


Figure 3.5: R after the execution of r6

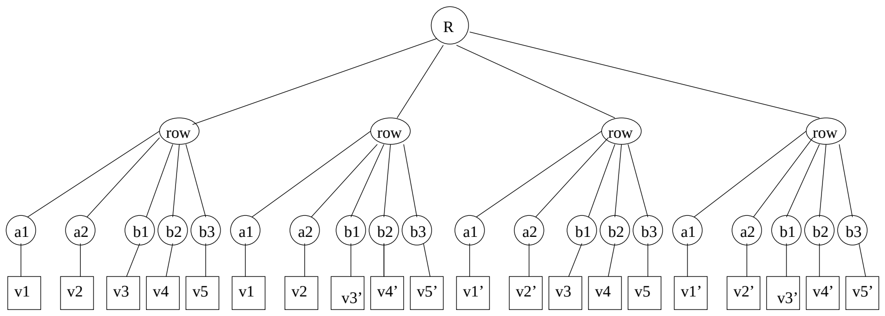


Figure 3.6: R after the execution of `r7`

This corresponds to the condition of the *while* loop that checks if the relation $R$ contains any rows or not. If $R$ has rows, rule `r1` fires leading to the triggering of rule `r3` that results in the execution of the loop body, indicated by `s` below. `s` starts by inserting the appropriate flag to initiate the first statement of the loop body. For example, if the first statement of the loop body is $R := R1 \cup R2$ then `s` will consist of the following action:

```
INSERT <union_flag_i/> document('temp.xml')/flags
```

The re-insertion of the `flag_while_i` at the end of the action part of `r1` leads to the repetition of the loop. If $R$ has no rows, the action part of `r2` deletes `flag_while_i` flag and the execution terminates:

```
r1:
ON INSERT document('temp.xml')/flags/while_flag_i
IF document('db.xml')/db/R/row
DO DELETE document('temp.xml')/flags/flag_while_i;
   INSERT <while_flag_i+1/> document('temp.xml')/flags;
   INSERT <while_flag_i/> document('temp.xml')/flags


r2:
ON INSERT document('temp.xml')/while_flag_i
```

```
IF not document('db.xml')/db/R/row
DO DELETE document('temp.xml')/while_flag_i

r3:
ON INSERT document('temp.xml')/while_flag_i+1
IF TRUE
DO s;
   DELETE document('temp.xml')/while_flag_i+1;
```

Up to this point, we have shown that XTL is able to emulate any *while* program and thus is at least as expressive as the *while* language over relational data. In the following paragraphs we describe the extensions that are necessary for XTL to also emulate $while_{new}$ programs and so become relationally query complete.

Emulating the instruction $R := new(S)$ of $while_{new}$ corresponds to the creation of a new XML element node `R` that has the same set of child elements under each `row` node as `S`, plus an extra child as the last sibling of the elements under each `row` node, representing the new relational attribute generated and holding the unique value generated by *new*.

To achieve this, XTL needs to be equipped with two functions named `newElement()` and `newValue()`. `newElement()` generates a new XML element node with a new arbitrary tag name, while `newValue()` generates a new value that is unique in the whole XML document. Furthermore, XTL needs to be equipped with the ability to define variables using `LET` expressions in the action part of rules. This is necessary so as to be able to keep a reference to a value or to the results of an expression that should be evaluated only once during the processing of the whole action part of a rule.

Rules r1-r3 below show a possible way to emulate the $R := new(S)$ instruction using these proposed extensions of XTL. Similarly to the other rule sequences above, two more rules ensure that $R$ exists and is empty, and they are omitted. The insertion of a new_flag_i+1 element into temp.xml triggers rule r1 that on activation deletes all the row nodes under the S node. This deletion triggers rule r2 that captures the deleted row nodes in the $delta variable and upon activation re-inserts the row node instances both under the new R node and under S. Rule r3 emulates the expansion of $R$ with a new attribute and unique values. It is triggered by the insertion of the row instances performed by rule r2. Once activated, it calls the newElement() function to create a new tag name and binds the result to the $r variable. Then it inserts the element $r under each row of R. After each new element $r has been inserted, it also places under $r a new value as resulting from a call to newValue(). The insertion of intermediate flags new_flag_i+2 and new_flag_i+3 by rules r1 and r2 and the check for their existence in the condition parts of rules r2 and r3, guarantees that the rules will be activated in the correct order and that no other deletion or insertion of row elements under S or R will activate them:

```
r1:
ON INSERT document('temp.xml')/flags/new_flag_i+1
IF TRUE
DO INSERT <new_flag_i+2/> BELOW document(''temp.xml'')/flags
      AFTER TRUE ;
   DELETE document('db.xml')/db/S/row;
   DELETE document('temp.xml')/flags/new_flag_i+1;


r2:
ON DELETE document('db.xml')/db/S/row
```

```
IF document('temp.xml')/flags/new_flag_i+2
DO INSERT <new_flag_i+3/> BELOW document('temp.xml')/flags
      AFTER TRUE ;
   INSERT $delta BELOW document('db.xml')/db/R AFTER TRUE ;
   INSERT $delta BELOW document('db.xml')/db/S AFTER TRUE ;
   DELETE document('temp.xml')/flags/new_flag_i+2;

r3:
ON INSERT document('db.xml')/db/R/row
IF document('temp.xml')/flags/new_flag_i+3
DO LET $r = newElement() IN
   INSERT $r BELOW $delta AFTER TRUE ;
   INSERT newValue() BELOW $delta/$r AFTER TRUE;
   DELETE document('temp.xml')/flags/new_flag_i+3;
```

## 3.5 Active XQuery

*Active XQuery* [BBCC02] is another XML ECA language developed approximately at the same time as the XTL language described above.

### 3.5.1 Syntactic Comparison with XTL

Similar to XTL, Active XQuery adopts the SQL3 trigger syntactic paradigm, with rules consisting of an event, condition and action part. Both languages use XPath for expressing rule event parts, although Active XQuery uses full XPath instead of the simple XPath expressions supported by XTL. For rule condition parts, Active XQuery uses full XQuery instead of the simple XPath expressions of XTL. Simple XPath is easier to encode and analyse while, as we have seen in the previous section, it does not prevent XTL from expressing complex application logic.

For rule action parts, Active XQuery uses the extension of XQuery with the update features described in [TIHW01]. The use of full XQuery (or full XPath) is an advantage when document order matters since, via the use of the position operations of XQuery, an XML fragment can be placed precisely in any part of the XML document, e.g. as second child of a specified node. This is not possible with XTL due to the limitations of simple XPath and simple XQuery expressions.

Concerning the types of actions that are supported, Active XQuery allows the replacement of an XML fragment by another in a single update statement, instead of requiring a deletion followed by an insertion as in XTL. Active XQuery also supports the renaming of XML node names as a single update statement. Extending XTL to support both of these features would be straightforward.

Active XQuery supports both BEFORE and AFTER triggers. XTL supports only AFTER triggers but it could easily be extended to support BEFORE triggers too. Active XQuery supports set- and instance-oriented rules, via the FOR EACH NODE and FOR EACH STATEMENT keywords, while in XTL this is implicitly indicated via the use of the `$delta` variable: the presence of `$delta` in a condition or action part implies an instance-oriented rule, otherwise the rule is set-oriented.

The definition of variables that store the results of an XQuery expression evaluation is supported in Active XQuery using LET expressions whose scope covers both the condition and the action part of the rule. Again, although this feature is missing from XTL, it does not add any extra expressiveness to Active XQuery and could easily be added to XTL.

### 3.5.2 Semantic Comparison with XTL

Active XQuery supports syntactic triggering, unlike XTL which supports semantic triggering. Although Active XQuery theoretically supports both immediate and deferred coupling modes, only immediate has been implemented.

A distinct characteristic of Active XQuery is the way that update statements are handled. One of the design requirements of Active XQuery rule execution was that it should be as close as possible to the semantics of SQL3 triggers [BBCC02]. However, given the hierarchical structure of XML documents and the "bulk" nature of updates in XML, such semantics cannot be replicated directly. For example, an insert action on XML may involve the insertion of a large XML fragment while a deletion may drop a whole branch of the XML document tree. To overcome this problem, Active XQuery proposes two different strategies that decompose the original bulk update into a set of atomic updates.

The first update decomposition strategy, called *loosely binding semantics* (LBS), decomposes the bulk XML update into a sequence of smaller granularity atomic updates, such that each XML element that is affected by the original update is addressed by one update in the generated sequence. A second approach, called *tightly binding semantics* (TBS), transforms a bulk update statement to a single expanded statement that redefines the initial bulk update as a set of nested updates to be performed in a specified order. The LBS approach is supported by the Active XQuery system implementation.

During the processing of the update sequence generated by LBS, triggering events are computed for each individual update in the sequence, and the resulting rule triggering occurs immediately after the execution of the individual update. This produces a sequence of interleaved updates and rule executions. In contrast, in XTL, update statements are treated as atomic operations and rules are triggered only after completion of the entire update. This difference means that the two systems may produce different results over the same XML data for the same top-level update.

For example, consider the example XML tree in Figure 3.7, where circles represent XML elements and boxes text nodes. Suppose we have two ECA rules `r_1` and `r_2` defined in XTL and in Active XQuery[4]:

```
r_1:
ON INSERT document('doc.xml')/a/b/c/f
IF $delta/../d/e
DO DELETE $delta/../d/e ;
   INSERT <e>$delta/text()</e> BELOW $delta/../d AFTER TRUE ;


r_2:
ON INSERT document('doc.xml')/a/b/c/d/e
IF $delta/../../g
DO DELETE $delta/../../g
```

The first rule `r_1` in response to an insertion of an `f` node, and given that there is at least one `e` node under `d` in the current branch under `c`, sets the text values of all the `e` nodes equal to that of the `f` node. The second rule `r_2` in response to the insertion of an `e` node deletes the node `g`, if it exists. We assume the `r_1` has a higher priority than `r_2`. Suppose now that a top-level update inserts a whole new `c` subtree, such as the subtree rooted at the leftmost `c` in Figure 3.7, under `b`. Let us see how this top-level update is treated by both of the ECA processing systems.

In XTL, the new XML fragment will be inserted under `b` in a single atomic update. After that, the triggered rules will be determined. Assuming the rules

[4]For brevity they are given here in XTL, but they can also be expressed similarly in Active XQuery.

described above are the only ones that are triggered, they will be processed one after the other according to their priorities, resulting in equal text values for all `e` nodes and in the deletion of `g`.

In Active XQuery the top-level update has different results. Before the update is processed, it is decomposed using the LBS algorithm into three smaller atomic updates, as illustrated by fragments 1 to 3 in Figure 3.7. After the execution of each of these three updates, the triggered rules are checked and processed as appropriate. So after the end of statement 2, it is found that the rule `r_1` is triggered due to the insertion of a new `f` node, but checking its condition it is found that no `e` node exists under `d` and so the rule's action part is not executed. The following execution of statement 3, will just delete the `g` node.

Thus, after the end of processing the same top-level update with the same set of triggered rules, different results are obtained by the two systems.

The update decomposition approach of Active XQuery does not seem to provide more expressive power than the XTL execution model because we can obtain the same result by explicitly using a sequence of finer-granularity actions. In contrast, the path expression duplication required to express all the individual updates in Active XQuery may impose a performance penalty that may require extra techniques, such as caching, to be applied in order to improve the overall system performance. A comparison of the relative performance trade-offs of XTL and Active XQuery was beyond the scope of this thesis but is an interesting area of possible future work.

## 3.6 Summary

In this chapter we have discussed the design of ECA languages in general and the specific issues concerning the design of such languages for XML. We reviewed a

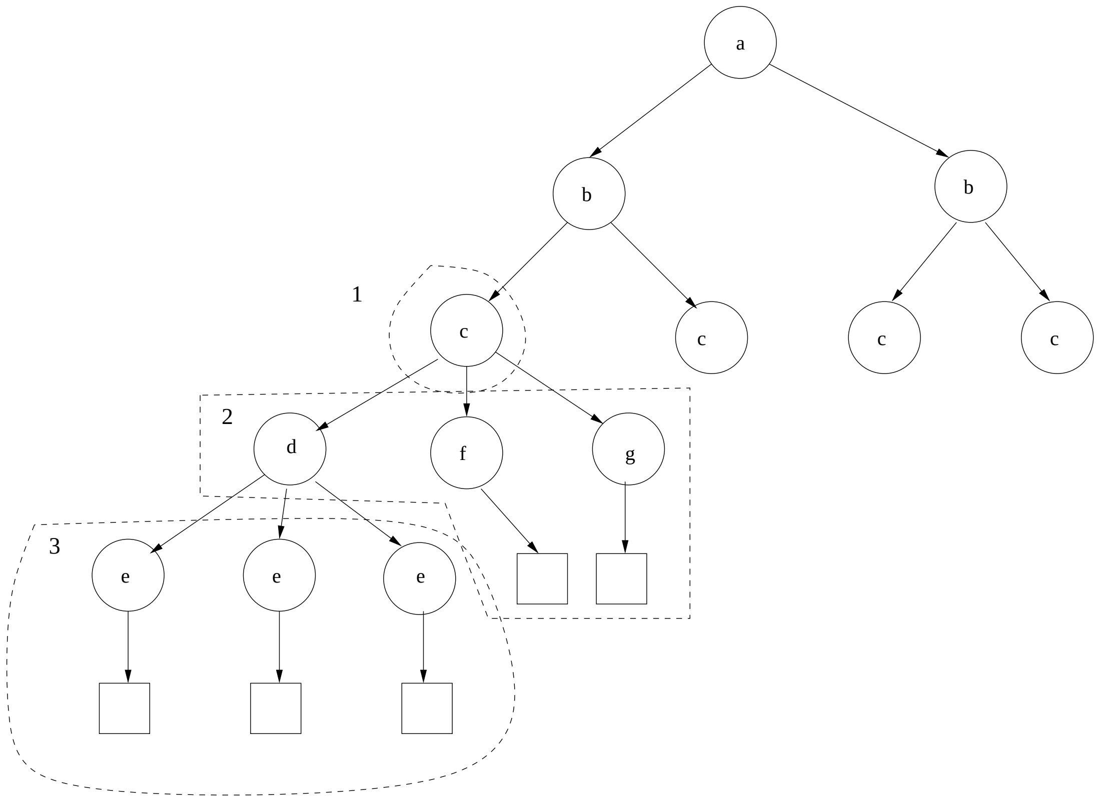


Figure 3.7: Example XML tree

specific XML ECA language, XTL, presenting its syntax and execution semantics. We then presented an analysis of the relational expressiveness of XTL and this is one of the key contributions of this chapter. It is also relatively straightforward to show that XTL can capture updates on relational data for an ordered database, following the results presented in [AV91]. For the future we plan a more detailed study that will investigate the expressiveness of XTL over relational data, as well as the precise update extensions that may be required for it to become update complete for relational data. We concluded the chapter with a comparison of XTL and Active XQuery, from both syntactic and semantic viewpoints.

In the following chapter we present the architecture and implementation of a prototype system that we have developed which supports the definition and execution of XTL rules.

# Chapter 4

# An XML ECA Rule Processing System

## 4.1 Introduction

Following the description of the syntax and execution semantics of the XTL language described in Chapter 3, we present in this chapter a prototype system we have developed that implements the XTL language and its execution semantics. The motivation for this implementation was to provide a proof of concept of the practical feasibility of the XTL language and its execution semantics.

The present chapter is organised as follows: in Section 4.2 we give a description of the system including its architecture and the components it comprises, followed by a discussion in Section 4.3 of the rule registration task. A performance study of the system is presented in Section 4.4. This study includes system modelling using analytical methods, and experimental results using both the analytical model and results from a set of experiments conducted on the actual system. In Section 4.5 an indexing structure for XML ECA rules is proposed in order to improve the time required to find rules that may be triggered by an event occurrence, and a study of the system performance using this indexing scheme with the analytical model is presented.

## 4.2 System Architecture

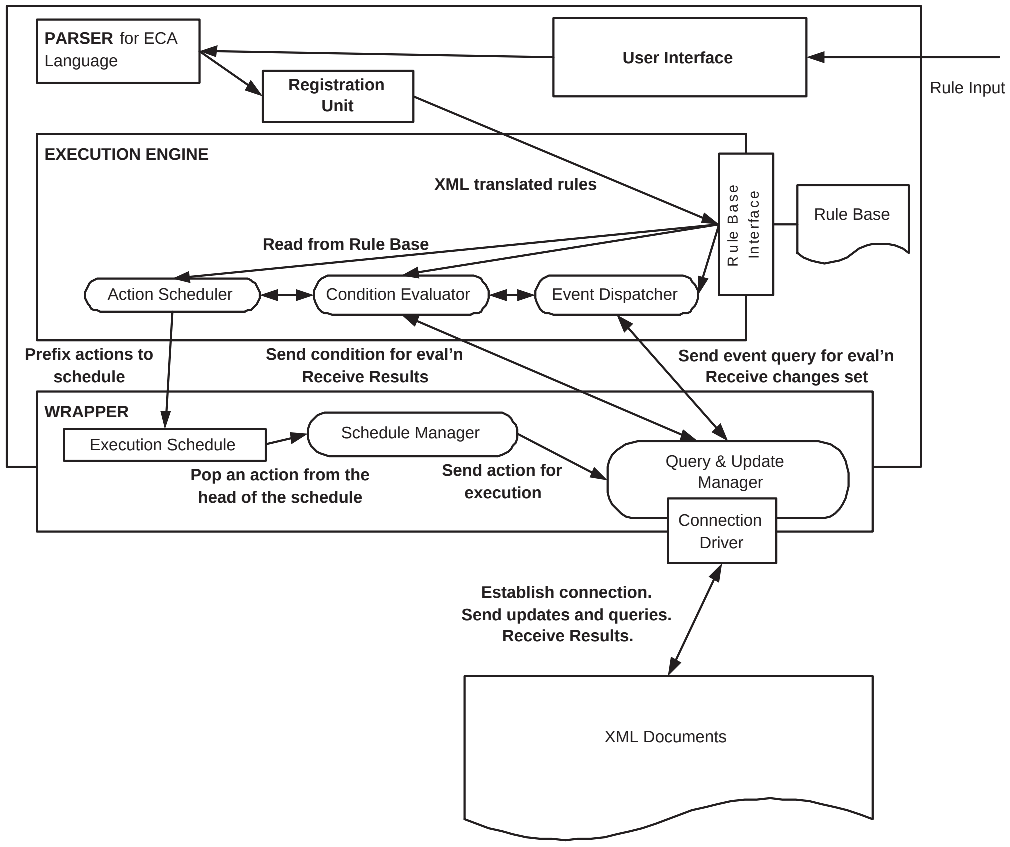


Figure 4.1: XML ECA Rule Processing System Architecture

The architecture of our system is illustrated in Figure 4.1.

The *Parser* parses and checks the syntactic validity of a new rule. For the construction of the parser, we have used the JavaCC [JavaCC05] lexer-parser generator. Valid rules are translated into an XML form and are added by the *Registration Unit* to the *Rule Base* (which is an XML file). Details about each rule are stored here, including its name, priority, and event, condition and action parts.

The *Execution Engine* encapsulates the rule processing functionality. In particular, the *Event Dispatcher*, *Condition Evaluator* and *Action Scheduler* implement these aspects of the rule processing, as we describe in more detail below. All of these components interface with the Wrapper in order to send and receive data to and from the underlying XML files. Due to the immaturity of the existing XML repository products in supporting a sufficiently expressive update language at the time that this first prototype was implemented (2002), we have used flat files and have exploited the functionality provided by DOM [W3CDOM] for interacting with them.

The *Execution Schedule* contains a sequence of *updates* — these have the same form as rule actions except that they do not contain any $delta expressions within them and in their place contain a function call whose argument is a linked list containing a set of node references for the `$delta` variable. By "$delta expression" we mean an XPath expression that contains a reference to the `$delta` variable.

The *Wrapper* interfaces with the XML files on disk. All update and query requests from the upper levels of the system pass through this component, which coordinates them. It undertakes to open files, submit queries and updates, and receive back results from them through the *Query & Update Manager*. All queries are performed directly by using XPath. For deletions, the set of nodes that will be deleted are identified by evaluating the XPath expression within the `DELETE` part of the request, and then all the subdocuments rooted at the nodes identified are deleted. For insertions, the set of nodes that will be affected are identified by evaluating the XPath expression within the `BELOW` part of the request, and then the fragment specified within the `INSERT` part is added as a new child of each of the nodes identified, placed relative to the existing children according to the

`AFTER` or `BEFORE` qualifier.

As well as performing queries and updates over XML data sources, the *Query & Update Manager* also has the task of notifying the *ECA Rules Processing Engine* of the occurrence of update events.

Rule execution begins with a request from the *Schedule Manager* to the *Query & Update Manager* to execute the update currently at the head of the schedule. In the case of an insertion, the Query & Update Manager executes the update and annotates the newly inserted nodes as “new”, while in the case of a deletion it annotates the nodes to be deleted as “to be deleted” without executing the deletion yet. Here we note that the annotation does not affect the physical representation of the file itself as its in-memory DOM encoding is used to perform the annotation. Annotations are removed before the file is serialised back to disk.

Following the execution of the update the Query & Update Manager notifies the Execution Engine of the event and details of the event including its type and the XML file it affects. Control then passes to the *Event Dispatcher*. This queries the Rule Base to obtain the rules that may be triggered by the event and then requests the *Query & Update Manager* to evaluate the XPath query of the event part of each rule that may be triggered by the update that was just executed. For each rule whose event query result set contains annotated nodes (either newly inserted or about-to-be deleted), the Event Dispatcher creates a `changes` set containing these annotated nodes, and the rule is triggered.

Control then passes to the *Condition Evaluator* which requests the Query & Update Manager to evaluate the condition part of each triggered rule on the affected document. Since a condition is generally a boolean expression with conjunctions, disjunctions and negations, each conditional is evaluated separately to determine if it is True or False, before a value for the whole boolean expression is determined by the *Condition Evaluator*. As the evaluation context is used either the root node, if there are no occurrences of `$delta` within a query, or otherwise each instance of the `changes` set. The rule's `delta` set is thus created, consisting of those members of its `changes` set for which the condition evaluates to true. If the `delta` set is non-empty, the rule fires and control is passed to the Action Scheduler to further process the rule. Otherwise, processing of this rule ends.

The *Action Scheduler* reformulates a given rule's action(s) in order to eliminate any instances of `$delta` expressions within them. The reformulation algorithm performs the following steps for each node within the rule's `delta` set: replaces the `$delta` variable in each of the $delta expressions by the current node of the `delta` set; evaluates each of the modified $delta expressions with respect to the updated document; and replaces each $delta expression within the rule's action(s) by the corresponding result of the previous step. The outcome of this reformulation is that one instance of the rule's action(s) is created for each node in the rule's `delta` set. These updates are now prefixed, in an arbitrary order, to the front of the schedule, since we employ immediate scheduling. If multiple rules have fired as a result of the last update executed, then the updates that result from their actions are prefixed the schedule in order of the rules' specified priorities. If the last update executed by the Query & Update Manager was a `DELETE`, then the actual deletion of the annotated nodes is now performed. If the last update was an INSERT then the new data are now deannotated. Control then passes once more to the Schedule Manager and the cycle repeats.

Our system architecture is modular. Each component is designed to be as independent from the other components as possible. This adds flexibility to the system, simplifying future improvements, extensions and modifications. For example, the implementation of a component could be replaced by a new improved one, or be moved onto another machine by being replaced by a Web Service call. The latter could be useful in the case that a distributed version of the system is required by an application.

## 4.3 Rule Registration

Rule registration begins with a user submitting a rule to the *Registration Unit* component (see Figure 4.1). The *Registration Unit* initialises the *Parser* to check the syntactic validity of the submitted rule and build its corresponding syntax tree. If the rule is syntactically correct, the syntax tree is passed back to the *Registration Unit*. For each part of the rule described in the syntax tree the *Registration Unit* creates its XML representation and then appends the rule to the Rule Base.

The Rule Base is represented by an XML file with document element `RuleBase`. Each rule is represented by a `Rule` element that is a child of `RuleBase`. The three rule parts are represented as child elements of `Rule` named `EventPart`, `ConditionPart` and `ActionPart`, respectively. The contents of each rule part are also encoded as child elements of the corresponding part element, e.g. `EventType` and `Expression`, containing the actual contents in a *Text* or *CDATA* section. For example, consider the rule $r_2$ given in Section 3.3, that upon insertion of a new `price` node checks if its value is greater than the current day's high and if so replaces it. The form of this rule as registered in the Rule Base is as follows:

```
<RuleBase>
    ...
    <Rule name="ReplaceHigh" priority="1148927618000">
        <EventPart>
            <Event>
                <EventType>INSERT</EventType>
                <FileName>shares.xml</FileName>
                <Expression>
                    <![CDATA[shares/share/day-info/high]]>
                </Expression>
            </Event>
        </EventPart>
        <ConditionPart>
            <ConditionsExpression>
                <condition>
                    <FileName>shares.xml</FileName>
                    <Expression>
                    <![CDATA[$delta >
                     $delta/../../month-info
                       [@month=$delta/../@month]/high]]>
                    </Expression>
                </condition>
            </ConditionsExpression>
        </ConditionPart>
        <ActionPart>
            <action>
                <ActionType>DELETE</ActionType>
                <what>
                    <filename>shares.xml</filename>
                   <Expression>
                   <![CDATA[
                    $delta/../../month-info/high]
                   ]>
                   </Expression>
```

```
                </what>
            </action>
            <action>
                <ActionType>INSERT</ActionType>
                <what>
                    <Expression><!CDATA[$delta]]></Expression>
                </what>
                <below>
                    <filename>shares.xml</filename>
                    <Expression>
                      <![CDATA[$delta/../../
                       month-info[@month=$delta/../@month]]
                      ]>
                    </Expression>
                </below>
                <before>TRUE</before>
            </action>
        </ActionPart>
    </Rule>
    ...
</RuleBase>
```

We see from the above that the name and the priority of the rule are added as attributes in the `Rule` element. The `EventPart` element consists of one `Event` element that in turn contains the contents of the event part of the rule: the event type under the `EventType` element; the name of the file on which the event occurs under the `FileName` element; and the XPath expression in a `CDATA` section under an `Expression` element. The `ConditionPart` element holds under the `ConditionsExpression` element a `condition` child element that contains the XPath expression of the condition and the XML file this is to be applied on. The `ActionPart` element has as children a set of `action` elements that correspond to the sequence of actions, in the order that they appear in the action part of the rule. Each `action` element contains the type of the action in an `ActionType` element. If the type is DELETE the `what` element contains the filename and the XPath expression specifying the data to be deleted. If the type is INSERT, the `what` element contains the data to be inserted, while its siblings named `below` and `before` or `after` correspond respectively to the `BELOW` and `BEFORE` or `AFTER` parts of the INSERT action.

Finally, we have implemented time-based rule priority in the sense that the earlier a rule registered the higher its priority is. So each rule that is submitted for registration is assigned a unique priority number that corresponds to the system's time (in milliseconds) when the registration process began.

## 4.4 Performance Study

In this research we have focused on *update response time* as the main performance criterion of the XML ECA system described above, defined as the mean time taken to complete all rule execution resulting from a single update submitted by a top-level transaction. Other choices of performance criteria could have been various types of system resource consumption, and this is an area of future research.

In the present section, we conduct a performance study of our system using analytical methods and also the system itself. In order to simplify the system model, we make a series of assumptions in Section 4.4.1. The main system modelling components are then presented in Section 4.4.2 before we proceed in Section 4.4.3 to the description of our analytical model for update response time. In Section 4.4.4 we present experimental results from both the analytical model and the system itself.

Previous work on modelling and performance evaluation of active rule processing systems has concentrated mostly on benchmarking and simulation techniques rather than on analytical methods. For example, the BEAST benchmark is employed in [GGD95] in order to evaluate the performance of event detection, rule management and rule execution in SAMOS [GD92]. Tests are defined for each component focusing on the time required to complete a processing step, detect events of various types, retrieve the correct rule from the rule base and execute rules in various coupling modes.

A set of simulation experiments are performed in [BB97] in order to evaluate the performance trade-offs for different rule execution semantics. The average transaction response time, defined as the average time elapsed from the transaction's arrival at the execution queue to its successful completion, is used as the main performance measure. A time similar to this, is the main performance measure in our study here.

## 4.4.1 Rule Triggering Assumptions for the Analytical Model

In our experience, the level of ECA rule triggering in database applications tends to be shallow, in that the probability of having $k$ levels of triggering in response to some top-level update decreases substantially as $k$ increases. We regard level 0 as the top-level update. Updates occurring at level $i+1$ result from rules that have been fired by updates occurring at level $i$. $p_{fire}(i)$ denotes the probability that an update occurring at level $i$ of rule execution causes a given rule to fire. This probability depends on the probability $p_{mt}$ that a given rule may be triggered by an update, the probability $p_t(i)$ that a rule that may be triggered is actually triggered at level $i$, and the probability $p_f$ that a rule that has been triggered actually fires. $p_{fire}(i)$ is given by the following equation:

$$p_{fire}(i) = p_{mt} \cdot p_t(i) \cdot p_f \tag{4.1}$$

We assume that the triggered probability $p_t(i)$ follows a geometric distribution with a constant reduction factor of $p_{reduct}$ at each level, while $p_{mt}$ and $p_f$ remain constant regardless of the triggering level[1]. So $p_t(i)$ is given by the following equation:

$$p_t(i) = p_t \cdot p_{reduct}^i \tag{4.2}$$

where $p_t$ denotes the probability that a rule is actually triggered as a result of an update.

As a consequence of equations 7.1 and 7.2:

$$p_{fire}(i) = p_{mt} \cdot p_t \cdot p_f \cdot p_{reduct}^i \tag{4.3}$$

implying that $p_{fire}$ also reduces geometrically with the level $i$.

Given that we currently support two types of events (INSERT and DELETE), we assume that the event parts of the rules in the Rule Base refer equally to these event types, so that half the rules may be triggered by an INSERT and the other half by a DELETE. So since in our current system implementation half of the rules "may-be-triggered", we set $p_{mt}$ equal to 0.5 in the analytical model. Thus, at each level $i$, the number of rules that may be triggered by a given update is

[1]Investigation of the effects of different probability distributions would be an interesting area of future work, as well as empirical experimentation with real sets of ECA rules.

given by

$$r_{mt} = p_{mt} \cdot n_{rules} \tag{4.4}$$

where $n_{rules}$ represents the number of rules in the Rule Base. $r_{mt}$ is a significant factor in the analytical model as the number of the rules that may be triggered corresponds to the number of event path expressions that need to be evaluated on the XML data in order to determine the rules that will actually trigger; this is a factor that, as shown below, has a major effect on the overall system performance.

As a consequence, the number of rules that are actually triggered at level $i$ by an update in level $i-1$ is given by the following equation:

$$r_t(i) = p_t(i) \cdot r_{mt} = p_{mt} \cdot p_t \cdot p^i_{reduct} \cdot n_{rules} \tag{4.5}$$

Similarly, the number of rules $r_{fire}(i)$ that fire for each event that occurs at level $i$ is given by

$$\begin{aligned} r_{fire}(i) &= p_{fire}(i) \cdot n_{rules} \\ &= p_{mt} \cdot p_t(i) \cdot p_f \cdot n_{rules} = r_t(i) \cdot p_f \end{aligned} \tag{4.6}$$

### 4.4.2 Rule Execution Modelling

Transactions consisting of queries and updates are submitted by applications to the system. Updates can cause rules to fire that may, in turn, cause the firing of further rules, increasing the system load. Queries submitted by a top-level transaction, although they do not cause any rule firing, can also increase the system load. Queries submitted during rule processing, for event or condition evaluation, add an extra factor, further increasing the system load.

In our analytical modelling of rule execution we assume two kinds of queues:

a *transaction queue* that accepts queries or updates arising from rule execution or from top-level transactions, and an *action scheduler queue* that queues updates resulting from rule firing which are then dispatched for execution to the transaction queue. For the transaction queue we assume a FCFS (First Come First Served) service discipline, while for the action scheduler queue, due to XTL's immediate rule coupling mode, we assume a LCFS (Last Come First Served) service discipline. Referring to the system architecture diagram in Figure 4.1, the *action scheduler queue* corresponds to the *Execution Schedule* in the *Wrapper* component, while the *transaction queue* is the query and update execution queue managed by the *Query & Update Manager*.

The query/update arrival rate at both queues is modelled as a Poisson process, which means that the arrival of a new item does not depend on any previous item (the process is memoryless) and the inter-arrival time is exponentially distributed. The exponential distribution of inter-arrival time leads to the *service time* also following an exponential distribution. By service time we mean the time for a query to be evaluated or an update to be executed over the XML data. An empirical justification of the service time's exponential distribution is given in [NJ00] and is as follows: The time required for a query/update depends on the number of data items accessed. Queries/updates that access a small number of data items are more frequent than those accessing a large number of data items. This leads to a geometrically distributed number of data items accessed per query/update. So the service time can be assumed exponential, which is the continuous version of geometrical.

Using Kendall's notation [Jai91], the transaction queues and the action scheduler queues are queues of the form $M/M/1$, where $M$ indicates exponential distribution for both the process arrival rate and the service time, and 1 specifies that the system provides a single service point.

### 4.4.3 Modelling Update Response Time

The update response time is the mean time taken to complete all rule execution resulting from a single top-level update submitted to the system. This update response time, $\overline{R}_{update}$, can be decomposed as follows:

$$\overline{R}_{update} = \overline{R}_{event} + \overline{R}_{cond} + \overline{R}_{action} \tag{4.7}$$

where $\overline{R}_{event}$ is the mean time taken for all event processing, $\overline{R}_{cond}$ the mean time taken for all condition processing and $\overline{R}_{action}$ the mean time taken for all action processing during the rule execution following a top-level update. We now consider each of these three components in turn.

**Event Response Time ($\overline{R}_{event}$)**

For each level of triggering $i$, the mean time taken to process the event part of a single rule at level $i$ is denoted by $\overline{R}^{rule}_{event}(i)$. This time is the mean time required to evaluate the event path expression of any rule in the Rule Base that has the same event type (INSERT or DELETE) and refers to the same XML file. For simplicity, henceforth we assume that the XML data are stored in one single XML file. The mean event processing time of a single rule at level $i$ is given by the following equation:

$$\overline{R}^{rule}_{event}(i) = t^{query}_{total}(i) \tag{4.8}$$

where $t^{total}_{query}(i)$ represents the mean time needed to perform an XPath query on the XML data. This time is equal to the sum of the mean time $t_q$ spent on evaluating the actual query plus the mean time $\overline{W}_{tq}(i)$ spent on waiting in the transaction queue:

$$t^{query}_{total}(i) = t_q + \overline{W}_{tq}(i) \tag{4.9}$$

The mean waiting time in the transaction queue, which follows the $M/M/1$ queue model, is [Jai91]:

$$\overline{W}_{tq}(i) = \frac{\lambda_{total}(i) \cdot t_q^2}{1 - \lambda_{total}(i) \cdot t_q} \tag{4.10}$$

Here $\lambda_{total}(i)$ represents the *total query or update request arrival rate* in the transaction queue which equals the sum of the arrival rate of query requests, $\lambda_{total}^{query}(i)$, and the arrival rate of update requests, $\lambda_{total}^{upd}(i)$, both at level $i$ of triggering:

$$\lambda_{total}(i) = \lambda_{total}^{query}(i) + \lambda_{total}^{upd}(i) \tag{4.11}$$

Here, $\lambda_{total}^{upd}(i)$ is given by:

$$\lambda_{total}^{upd}(i) \quad = \quad \lambda_{ext}^{upd} + \lambda_{int}^{upd}(i) \tag{4.12}$$

where $\lambda_{ext}^{upd}$ is the arrival rate of top-level updates that are submitted from outside the rule processing system, and $\lambda_{int}^{upd}(i)$ is the arrival rate of updates that are generated as a result of rule firing at level $i$.

Each of the externally arriving top-level updates will cause a first level of triggering. Each of the updates contained within the transactions generated as a result of this rule triggering may cause further rules to fire, causing more transaction traffic to be generated, and so on. So the total arrival rate of updates at level $i$ of triggering can be expressed as:

$$\begin{aligned}
\lambda_{total}^{upd}(i) &= \lambda_{ext}^{upd} + \lambda_{ext}^{upd} \cdot r_{fire}(1) \cdot N_{action} \\
&+ (\lambda_{ext}^{upd} \cdot r_{fire}(1)) \cdot r_{fire}(2) \cdot N_{action}^{2} \\
&+ \ldots \\
&+ (\lambda_{ext}^{upd} \cdot r_{fire}(1) \ldots \cdot r_{fire}(i-1)) \cdot r_{fire}(i) \cdot N_{action}^{i} \\
&= \lambda_{ext}^{upd} \cdot [1 + \sum_{j=1}^{i} N_{action}^{j} \cdot \prod_{l=1}^{j} r_{fire}(l)] \qquad (4.13)
\end{aligned}$$

where $r_{fire}(l)$ is the number of rules that fire at triggering level $l$, and $N_{action}$ is the total number of individual updates each of the $r_{fire}(l)$ rules generates.

According to equations 4.1, 4.3, 4.6 and 4.13, the update arrival rate at level $i$ is

$$\lambda_{total}^{upd}(i) = \lambda_{ext}^{upd} \cdot (1 + n_{rules} \cdot p_{mt} \cdot p_{t} \cdot p_{f} \cdot \sum_{j=1}^{i} N_{action}^{j} \cdot \prod_{l=1}^{j} p_{reduct}^{l})$$

In 4.11, $\lambda_{total}^{query}(i)$ equals the sum of $\lambda_{ext}^{query}$ plus $\lambda_{int}^{query}(i)$, where $\lambda_{ext}^{query}$ is the arrival rate of queries that are submitted from outside the rule processing system and $\lambda_{int}^{query}(i)$ is the arrival rate of queries that are generated as a result of rule processing at level $i$, i.e. event and condition expression evaluation:

$$\lambda_{total}^{query}(i) = \lambda_{ext}^{query} + \lambda_{int}^{query}(i) \qquad (4.14)$$

So the total query arrival rate at level $i$ of triggering can be expressed as:

$$\begin{aligned}
\lambda_{int}^{query}(i) &= \lambda_{ext}^{query} + \lambda_{total}^{upd}(0) \cdot (r_{mt} + r_t(1) \cdot N_{cond}) \\
&+ \lambda_{total}^{upd}(1) \cdot (r_{mt} + r_t(2) \cdot N_{cond}) \\
&+ \ ... \\
&+ \lambda_{total}^{upd}(i-1) \cdot (r_{mt} + r_t(i) \cdot N_{cond}) \\
&= \lambda_{ext}^{query} + \sum_{j=1}^{i} \lambda_{total}^{upd}(j-1) \cdot (r_{mt} + r_t(j) \cdot N_{cond}) \qquad (4.15)
\end{aligned}$$

The factor $r_t(j) \cdot N_{cond}$ gives the number of condition expressions evaluated per triggering level as $r_t(j)$ gives the number of rules triggered at triggering level $j$ and $N_{cond}$ the mean number of condition expressions each of the triggered rules contributes (see below).

Assuming that the action part of each rule contributes $N_{action}$ updates to an action schedule (see below), then substituting equation 4.9 into equation 4.8, and summing over all $k$ triggering levels, we obtain the mean time taken to process all event queries over all $k$ triggering levels during the rule execution following a top-level update as:

$$\overline{R}_{event} = \sum_{i=1}^{k} \{ r_{fire}(i-1) \cdot N_{action} \cdot r_{mt} \cdot \overline{R}_{event}^{rule}(i) \} \qquad (4.16)$$

where the factor $r_{fire}(i-1) \cdot N_{action}$ represents the total number of individual updates caused by the previous level of triggering and $r_{mt}$ represents the number of rules that have event part with the same event type as the event that occurred.

### Condition Response Time ($\overline{R}_{cond}$)

Only a portion of the rules stored in the Rule Base will actually be triggered. According to the assumptions made in Section 4.4.1, for triggering level $i$ this is equal to $r_t(i) = r_{mt}(i) \cdot p_t$, where $p_t$ is the probability that a may-be-triggered rule is actually triggered. The time needed the condition part of a triggered rule to be evaluated is given by the following equation:

$$\overline{R}^{rule}_{cond}(i) = t^{total}_{query}(i) \cdot N_{cond} \tag{4.17}$$

where $N_{cond}$ represents the total number of expression evaluations required to evaluate the condition part of a rule, assuming that a rule has an average $\ell^{cond}_{inst}$ condition part instances (due to `$delta`) and each of them contains $\zeta$ path expressions, $N_{cond} = \ell^{cond}_{inst} \cdot \zeta$.

So, the mean time taken to process all condition queries over all $k$ triggering levels during the rule execution following a top-level update is as follows, recalling that $r_t(i)$ rules are triggered at the $i$th triggering level:

$$\overline{R}_{cond} = \sum_{i=0}^{k} \left( r_t(i) \cdot \overline{R}^{rule}_{cond}(i) \right) \tag{4.18}$$

### Action Response Time ($\overline{R}_{action}$)

The number $r_{fire}(i)$ of the rules that fire at the $i$th level of triggering, is expressed as the product of the probability $p_f$ that a triggered rule fires and the number $r_t(i)$ of rules that were triggered at level $i$, $r_{fire}(i) = r_t(i) \cdot p_f$. The mean time needed for the execution of the action part of each of the rules that fire is given below:

$$\overline{R}^{rule}_{action}(i) = t^{total}_{action}(i) \cdot N_{action} \tag{4.19}$$

where $N_{action}$ is the average number of updates placed by a rule to the *action scheduler queue* and $t^{total}_{action}$ is the mean time taken for an update to be executed in the system. We assume that there are an average of $\ell_{inst}$ instances contributed by an action, and that each rule's action part contains on average $size_{ap}$ individual actions. So the number of updates, $N_{action}$, placed by a rule to the action scheduling queue is:

$$N_{action} = \ell_{inst} \cdot size_{ap}$$

After an instance of the action part is placed on the action scheduling queue at level $i$, each update within it waits for a time of $\overline{W}_{upd}(i)$ before being sent to the transaction queue, where it will also wait for $\overline{W}_{tq}(i)$ time before being executed over the XML data, in time $t_{srv}$. Thus, the mean time spent on the execution of an update within an instance of an action part is:

$$t^{total}_{action}(i) \quad = \quad t_{srv} + \overline{W}_{upd}(i) + \overline{W}_{tq}(i) \tag{4.20}$$

where the $t_{srv}$ is assumed to be equal to the time needed for a query to be evaluated $t_q$ and $\overline{W}_{tq}(i)$ is the mean waiting time in the transaction queue.

The mean waiting time in the $M/M/1$ action scheduling queue at level $i$ of triggering is given by the following equation [Jai91]:

$$\overline{W}_{upd}(i) = \frac{\lambda^{upd}_{total}(i) \cdot t^2_{srv}}{1 - \lambda^{upd}_{total}(i) \cdot t_{srv}} \tag{4.21}$$

From the fact that $r_{fire}(i)$ rules fire at level $i$, and that there are $k$ triggering levels, we obtain the time taken for all rule action processing as

$$\overline{R}_{action} \quad = \quad \sum_{i=1}^{k} r_{fire}(i) \cdot \left(\overline{R}^{rule}_{action}(i)\right)$$

into which the right-hand sides of equations 4.19 and 4.6 can be substituted.

### 4.4.4 Experimental Results

As well as developing the analytical performance model described above, we have conducted a series of experiments on our actual system implementation. We present the experimental results from both the analytical performance model and the system measurements.

The actual system implementation was used to measure and calibrate the parameter $t_q$, as shown in Table 4.1. In the absence of information about real-world XML ECA rule sets we have also fixed the remaining parameters to the values shown in Table 4.1. For the experiments with the analytical model, we have assumed that no external transaction arrival occurs (i.e. $\lambda^{query}_{ext} = 0$ and $\lambda^{upd}_{ext} = 0$) except from the top-level update. The reason for this assumption was in order for the analytical model study to be as close as possible to the conditions under which the experiments on the real system were conducted, since in the latter case the only external transaction on the system is the top-level update that initiates the rule triggering.

#### Analytical Study Results

We examine the general performance trends of processing XTL rules using a system such as the one described in the previous sections. We consider the general trends as the primary result of our study rather than the actual values obtained.

| **Parameter** | **Base setting** |
|---|---|
| $t_q$ | 0.0008 sec |
| $k$ | 30 |
| $N_{cond}$ | 4 |
| $N_{action}$ | 8 |
| $size_{ap}$ | 4 |
| $p_{reduct}$ | 0.2 |
| $p_t$ | 0.3 |
| $p_f$ | 0.5 |
| $p_{mt}$ | 0.5 |

Table 4.1: Parameter Base Values

Altering values of the parameters in Table 4.1 affects the absolute values but not the general performance trends.

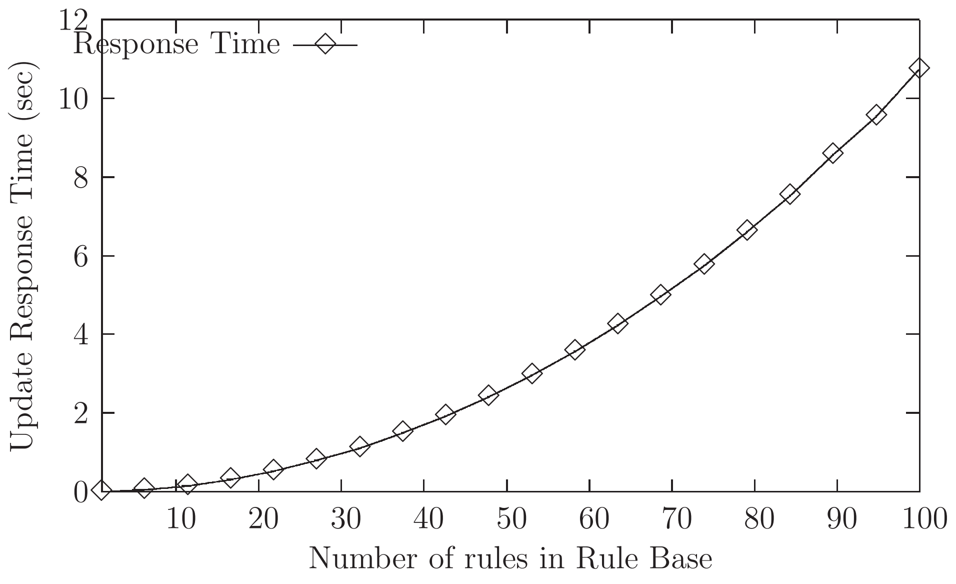


Figure 4.2: Analytical Performance Results

Figure 4.2 shows how the update response time varies as the number of rules in the rule base, $n_{rules}$, increases. From the shape of the curve it is clear that the system does not scale well. As $n_{rules}$ increases, the update response time tends to rise quite rapidly towards high values. Although we obtain relatively low times for few rules in the rule base, the rate of increase is non-linear and the response time soon reaches high values that would make the system unusable in a real application environment.

As $n_{rules}$ increases, the number of rules that will potentially be triggered and activated increases, as does the number of query and update requests placed in the *transaction queue*. Furthermore, the higher $n_{rules}$ is the greater the effects of triggering cascades become, resulting in even more query and update requests in the *transaction queue*.

The response time increases non-linearly for as long as the system is stable, i.e. so long as the arrival rate in the transaction queue is less than the rate at which requests that can be served. After the arrival rate exceeds the service rate, the system become unstable, which can be interpreted as the transaction queue growing uncontrollably large, so that it soon floods the memory of the system causing at first the system to slow down dramatically and then, potentially, the application to crash.

The reasons for this behaviour seem to be the following:

- All the query and update requests are served by a single transaction queue. As the transaction service time, $t_q$, remains constant while the number of query and update requests increases as greater number of rules need to be processed, the waiting time in the transaction queue increases dramatically until the arrival rate overwhelms the rate at which the requests can be served.

- The number of query evaluations needed to find the rules that are actually triggered. Each time that an event occurs we have to check all the rules that have the same event type in their event parts (INSERT or DELETE) regardless of the fragment of data affected by the update.

Altering the values of the performance variables in Table 4.1 simply shifts this in stability point (i.e. the point where the arrival rate equals the service rate) to a smaller or larger number of rules in the rule base, but without affecting the performance trends that are observed in all cases.

**System Experiment Results**

For the experiments with the actual system, we have used a fragment of the DBLP [DBLP06] XML database. Rules are generated randomly according to the value of parameters such as the proportion of instance and set oriented rules, the proportion of INSERT and DELETE events and actions, and the number of actions in the action part of a rule. Three different rule sets of 50 rules each were randomly generated and three separate experiments for each of these three sets of rules were performed.

In order to examine system scalability, the size of the rule base, $n_{rules}$ was increased by successively accumulating a replica of the initially generated 50 rules in the existing rule base. That is, the rule base was initially the original set of 50 rules, and then increased linearly to 100, 150, 200 etc. rules. Thus the amount of first-level rule triggering and activation also increased linearly. For all three rule sets, the same top-level update initialised the rule execution. Each experiment was performed four times for each different value of $n_{rules}$ and the average time taken. Due to limitations imposed by the current DOM-based implementation of our system, the maximum triggering level was set to 6. The experiments were conducted on a PC running Linux Fedora Core 4 featuring an AMD Athlon 64bit Dual Core processor and 2 Gb of RAM.

Figure 4.3 shows the results of the experiments for the three rule sets. The number of rules triggered per level for each of the three rule sets when the Rule

Base contains 50 rules is shown in Table 4.2.

| **Rule Set** | **Level 1** | **Level 2** | **Level 3** | **Level 4** | **Level 5** | **Level 6** |
|---|---|---|---|---|---|---|
| *RuleSet1* | 4 | 3 | 2 | 2 | 2 | 2 |
| *RuleSet2* | 3 | 3 | 2 | 2 | 2 | 1 |
| *RuleSet3* | 3 | 2 | 2 | 2 | 1 | 1 |

Table 4.2: Rules triggered per triggering level (for 50 rules in Rule Base)

From the graphs in Figure 4.3 we observe that the update response time for *Rule Set 1* increases in a non-linear fashion as $n_{rules}$ increases while for *Rule Set 2* and *3* an almost linear behaviour is observed as $n_{rules}$ increases. Although the observation about *Rule Set 1* agrees with the predictions made by the analytical model, *Rule Set 2* and *3* tend to diverge from this. An explanation for this might be the fact that for each rule set a different number of rules is triggered per triggering level (see Table 4.2). Due to the way the rule base increases, the number of rules triggered per level also increases by successively accumulating those that triggered when then size of the rule base was 50. So for example, when the size of rule base is 100, *Rule Set 1* has *8-6-4-4-4-4* rules triggered at the six levels, and so on. As the size of the rule base increases, the total amount of rules that need to be processed increases significantly. Since *Rule Set 2* and *3* have fewer rules to be processed per level, they give lower update response times and a behaviour closer to linear as $n_{rules}$ increases.

The different absolute update response times observed for different rules sets is due to the different number of rules triggered per rule level.

A comparison of Figure 4.3 with Figure 4.2 reveals a noticeable difference in the absolute values of the update response time. This is due to implementation choices made for the system and the assumptions made in the analytical model. In particular, our use of a DOM-based approach for querying and updating XML

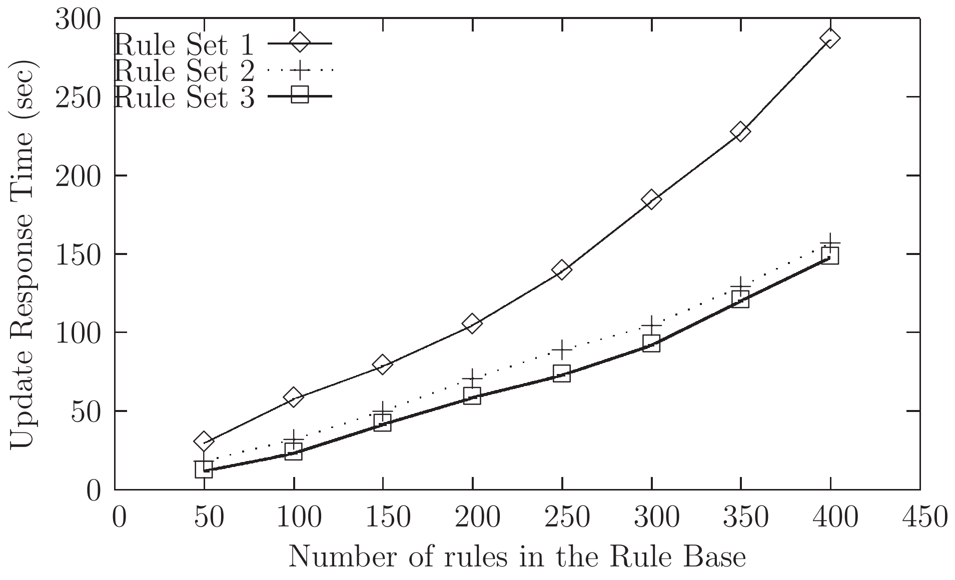


Figure 4.3: Experimental Results

documents in the system implementation requires that each time a query or update needs to be performed over an XML document, the document be encoded in DOM format and loaded as a whole into memory. This approach is extremely slow since both the rule base and the data are in XML format, and performing experiments for $n_{rules}$ higher than about 400 is infeasible. Also, the query time for both the data and the rule base depends heavily on the size of the XML document. The future use of an XML repository that directly supports query and update techniques would remove those limitations from our implementation.

## 4.5 Indexing XTL Rules

From equation 4.8 in the analytical model we recall that each time that an INSERT or DELETE occurs, the subset of rules in the rule base that refer to an INSERT or DELETE event needs to be checked to see if they are actually triggered.

In order to reduce the size of this may-be-triggered rule set and thus reduce the detection time for triggered rules, we now propose an indexing scheme for the event parts of rules and examine its impact on update response time.

The event part of each rule can be split into the type of the event, the XML document that is affected and the XPath expression specifying the part of the XML document that, if changed, will trigger the rule. We create an indexing scheme that exploits these characteristics to find the set of rules that are candidates for triggering. The index has the form of graph $G$ representing a non-deterministic finite automaton $G$. The edges of $G$ are labelled by the names of the files affected by the event part of the rules, the types of the events (INSERT or DELETE), that can be applied per file, and each step of an XPath expression appearing in the event part of a rule. For simplification purposes, in the current version of the indexing scheme we propose, the qualifiers are stripped out of the expressions. See Figure 4.5 for an example.

Starting from the initial state of $G$, the first two levels of edges in the automaton are labelled with the filenames and the event types. The rest of the automaton is built from the steps on the event path expression following the approach introduced in [GGM+04]:

- Each step in the expression is represented by an edge on the graph labelled with the step's name (i.e. element tag name or attribute name). In case of a `‘*’` element or a `’@*’` attribute, a `‘*’` or a `’@*’` is used as the edge label.

- The descendant axis is represented by an empty transition (labelled with $\epsilon$) to a state with a loop edge labelled with $*$.

Each state stores the set of the rules that have the node corresponding to this state as the output node of their event query and thus may be triggered, ordered by their relative priority.

When a new rule is submitted for registration to the system, it is indexed as follows:

I. Check the XML file in the event part. If there is an edge labelled with the current filename then follow it and proceed to the next step; if there is not, then create a new state and an edge, labelled with the filename, connecting the initial state and the newly created state; and then proceed to step II.

II. In a similar way for the event type, if the edge labelled by the event type of the rule exists, follow it and proceed to the next step, or otherwise create a new state and edge labelled with the event type and then proceed to step III.

III. If the expression contains any `parent` (`'..'`) step within the path, then transform it to the equivalent expression without parent steps using the technique described in [OMFB02]. [OMFB02] defines a set of XPath expression equivalences that are used to define an algorithm called *rare* (**r**everse **a**xis **re**moval) for rewriting XPath expressions eliminating any reverse axes. For example, using the *rare* algorithm the path expression `a//b/../d` is transformed to `a//.[b]/d` that contains no `parent` (`'..'`) steps. Since the technique described in [OMFB02] generates qualifiers during the transformation of the `parent` step, we strip these qualifiers out of the XPath expression, so that the expression `a//.[b]/d` becomes `a//./d` which is equivalent to `a//d`.

IV. Starting from the first step of the path expression, check if the same path exists already in the index, following one by one the path expression steps and the graph edges. If an edge labelled with the same name as the current step exists, then continue traversing the automaton. If no such an edge exist, then create a new edge labelled with the name of the step and a new state to connect it with. In case that between two steps in the expression a descendant axis appears then check if an $\epsilon$ transition edge and a $*$-labelled loop edge, exists already or otherwise create them before proceed to the next step. If the current step is the result node of the event XPath expression, then add the rule's name to the rule set of the state and terminate the traversal; otherwise we leave the set unchanged.

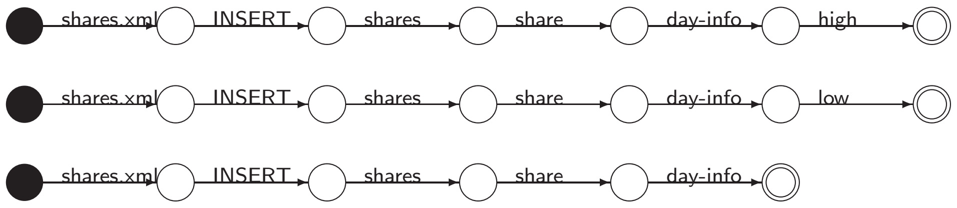
The insertion of a new rule has to check the file the event is referring to, the type of the event (INSERT or DELETE), and where each of the $n_{steps}$ number of steps of the event XPath expression will be placed within the $n$ nodes of the index subtree. This results in a worst case complexity of $O(n+n_{steps}+2) = O(n+n_{steps})$.

The occurrence of a top-level update $u_0$ notifies the system to start searching the index. Depending on $u_0$, removing `parent` (`‘..’`) steps and stripping out any qualifiers may be required. Furthermore, in the case of an insertion of a new XML fragment that is constructed via an XQuery expression, we may have to check the index for each of the paths created by the insertion. For example, in the following update expression:

```
INSERT <day-info day="01" month="05">
          <high>123.43</high>
          <low>112.53</low>
       </day-info>
  BELOW doc('shares.xml')/shares/share AFTER TRUE
```

we have to check the index for the existence of the `shares/share/day-info/high`, `shares/share/day-info/low` and `shares/share/day-info` paths in `shares.xml`.

Similarly to events, an update can be split into its type, the file it operates on and its path expression. From the update expression(s), we again form an automaton (or automata), following the same approach with the index, that will matched against the index. For example, for the update above, the following three automata will be generated and matched against the index:



In the automata above a black node corresponds to the initial state and a double-circled node to a final state.

The SEARCH_INDEX function shown in Figure 4.4 is used for matching path expressions corresponding to updates to those represented in the index. The function takes as arguments the current state $v$ in the index automaton and the current state $s$ in the update automaton. The function is initially called with the initial state of each automaton. The $rules : State \rightarrow RuleSet$ function returns the rule set contained in the specified state. The rules contained in the matched states are accumulated in the $RS$ variable which, by the end of the algorithm execution, contains the rules that may be triggered by the given update. $F$ is the set of finite states in the update automaton. The function $\delta_{upd} : State \rightarrow Label \rightarrow StateSet$ returns the set of states that result from all the transitions from the given state in the update automaton via an edge with a given label. The function $\delta_{ix} : State \rightarrow Label \rightarrow StateSet$ performs the same operation but over the index automaton.

**Example**: Consider the XML ECA rules $r_1$ and $r_2$ defined in Section 3.3.1. Consider also an XML ECA rule $r_3$ with the following event part:

```
1. SEARCH_INDEX $(v, s)$
2. begin
3.   if $s \in F$ then $RS \leftarrow RS \cup \{(rules(v))\}$
4.   for each $t \in \delta_{upd}(s, \epsilon)$ do SEARCH_INDEX$(v, t)$
5.   for each $w \in \delta_{ix}(v, \epsilon)$ do SEARCH_INDEX$(w, s)$
6.   for each $w \in \delta_{ix}(v, a)$ s.t. $a \neq \epsilon$ do
7.     if $a =$ '*' then
8.       for each $t \in \delta_{upd}(s, b)$ s.t. $b \neq \epsilon$ do SEARCH_INDEX$(w, t)$
9.     else
10.      for each $t \in (\delta_{upd}(s, a) \cup \delta_{upd}(s, *))$ s.t. $a \neq \epsilon$ do SEARCH_INDEX$(w, t)$
11. end
```

Figure 4.4: Index Searching Algorithm

```
ON INSERT document('shares.xml')/shares/share/day-info/@*
```

that triggers when a new attribute is inserted in the `day-info` element and another rule $r_4$ with event part:

```
ON INSERT document('shares.xml')/shares/share//price
```

Figure 4.5 illustrates how these four rules are indexed using the indexing scheme described above.

Now consider a top-level update $u_0$ that performs the following action which inserts a new `price` after the last price of any share.

```
INSERT <price time="10:32">143.3</price>
  BELOW document('shares.xml')/shares//prices
  AFTER TRUE
```

In order to find out which rules may be triggered by $u_0$, we consult the index structure shown in Figure 4.5.

We start applying the search algorithm of Figure 4.4 using $v$ and $s$ as indicated in Figure 4.5. Searching up to the states $v'$ and $s'$ is straightforward, as the

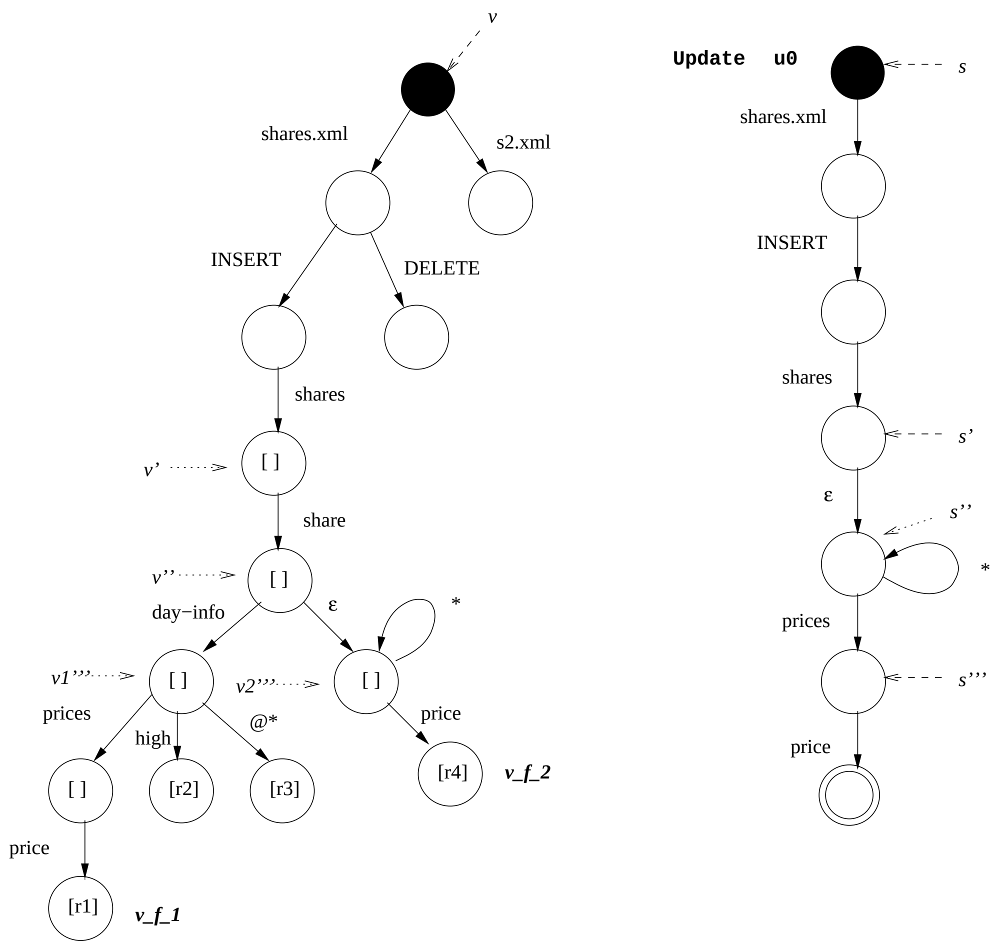


Figure 4.5: Example XTL rule index

same transition edges exist in both $u_0$ and the index. The next recursive call of SEARCH_INDEX with parameters $(v', s')$ executes line 4, due to the $\epsilon$ transition edge in $u_0$, causing another recursive call with parameters $(v', s'')$. In turn, the execution of the search with parameters $(v', s'')$ executes line 10 of the algorithm, where the $(v', v'')$ (i.e. *share*) edge matches with the loop '*' edge on $s'$, leading to another recursive call of SEARCH_INDEX with parameters $v''$ and $s''$. In the following step, we have a transition from $v''$ to $v1'''$, that matches again with the '*' loop on $s''$, calling in turn SEARCH_INDEX with $v1'''$ and $s''$ (line 10) that results in the final node of $u_0$ matching node $v\_f\_1$. Hence, rule $r_1$ is added to the set $RS$.

The algorithm now drawbacks to the execution of the function with parameters $v''$ and $s''$. The $\epsilon$ transition from $v''$ to $v2'''$ causes line 5 to execute calling again SEARCH_INDEX with parameters $v2'''$ and $s''$. The transitions from $v2'''$ are the *price* edge, that clearly does not lead to a match, and the * loop edge that brings the execution to line 8 of the algorithm and another recursive call of SEARCH_INDEX with parameters $v2'''$ and $s'''$, which results in matching the final state of $u_0$ with the state $v\_f\_2$.

So, from the accumulation in $RS$ of the rule sets stored in $v\_f\_1$ and $v\_f\_2$, we have that the rules $r_1$ and $r_4$ may-be-triggered in response to $u_0$.

### 4.5.1 Analytical Performance Model Using Index

If we employ the Rule Base indexing scheme described in Section 4.5 to find the rules in the Rule Base that may be triggered, the number of rule event parts that have to be evaluated on the XML data is decreased. Assuming an average time $t_{ix}$ is required to access the index, then the equation 4.16 becomes:

$$\overline{R}_{event} = \sum_{i=1}^{k} \{ r_{fire}(i-1) \cdot N_{action} \cdot (t_{ix} + r_{mt} \cdot \overline{R}^{rule}_{event}(i)) \} \qquad (4.22)$$

since for each update at level $i$ we spend $t_{ix}$ time to query the index plus the time needed to process the resulting rules. From equation 4.4, where $r_{mt} = p_{mt} \cdot n_{rules}$ the probability $p_{mt}$ that one of the $n_{rules}$ "may-be-triggered" is now $p_{mt} << 0.5$ since the index filters out a much larger portion of the $n_{rules}$ rules. $p_{mt}$ is a measure of the quality of the index as it specifies how "selective" this is.

The graphs shown in Figures 4.6, 4.7, 4.8, and 4.9 illustrate the system performance trends according to the analytical model if an indexing structure like the one described in Section 4.5 is used and for various values of $p_{mt}$. The system parameters, apart from $p_{mt}$, have been kept the same as in Table 4.1 and the average index access time $t_{ix}$ has been fixed at 0.01 seconds.

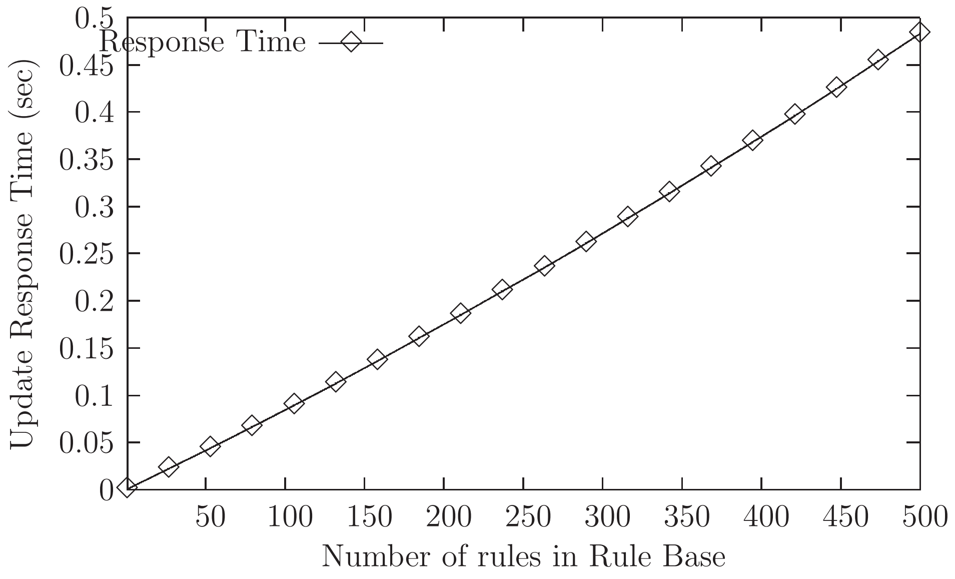


Figure 4.6: Analytical Performance Results — With Index ($p_{mt} = 0.01$)

Experimenting with a range of values for $p_{mt}$ starting from 0.01 to 0.4 we observe that with low $p_{mt}$ values (highly selective index) the system shows good scalability characteristics and the update response time increases linearly with the number of rules in the rule base, $n_{rules}$. This is because the mean time needed for the event part of a rule to be processed has been decreased compared to the case of no indexing, since only a portion of rules equal to $r_{mt}(i) = p_{mt}(0) \cdot p^{i}_{reduct} \cdot n_{rules} << n_{rules}$ needs to be checked. This means fewer queries are placed for execution on the transaction queue, an effect that becomes more significant in cascaded rule triggering. As the value of $p_{mt}$ increases (less selective index) the benefits of the index start to disappear and the system shows a non-linear behaviour that becomes progressively more marked as the value of $p_{mt}$ approaches 0.5, where in effect the system performs as if no index exists.

These results point to the use of a rule index as an effective way to significantly

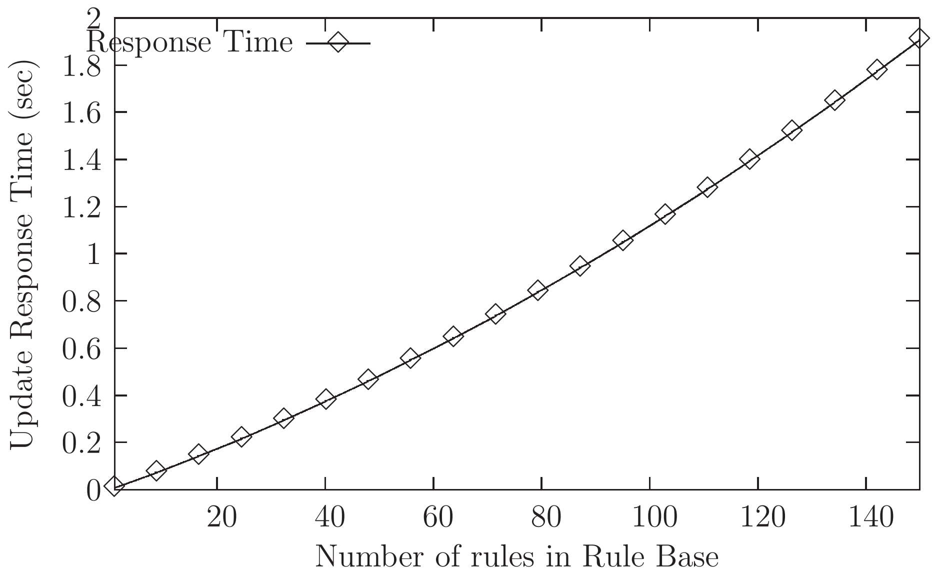


Figure 4.7: Analytical Performance Results — With Index ($p_{mt} = 0.1$)

improve system performance. Due to lack of time, no indexing scheme has been implemented so far in our current system, and thus no experiments using indexes with the actual system were possible. We leave as future work the development of a rule indexing mechanism for our system.

### 4.5.2 Comparison with a hard-coded approach

In the study above, we have presented the performance characteristics of our XTL ECA rule processing system. As we discussed in Chapter 1 of the thesis (Section 1.1), an alternative to using ECA rules for providing reactive functionality in an application would be to 'hard code' the reactive functionality into the application.

In the hard-coded case, each particular event that needs to be handled would be implemented directly in the application code. For example, referring to the XML file of Figure 2.1, a possible way to implement the requirement that the daily high price of a share should be updated in response to a new price insertion,

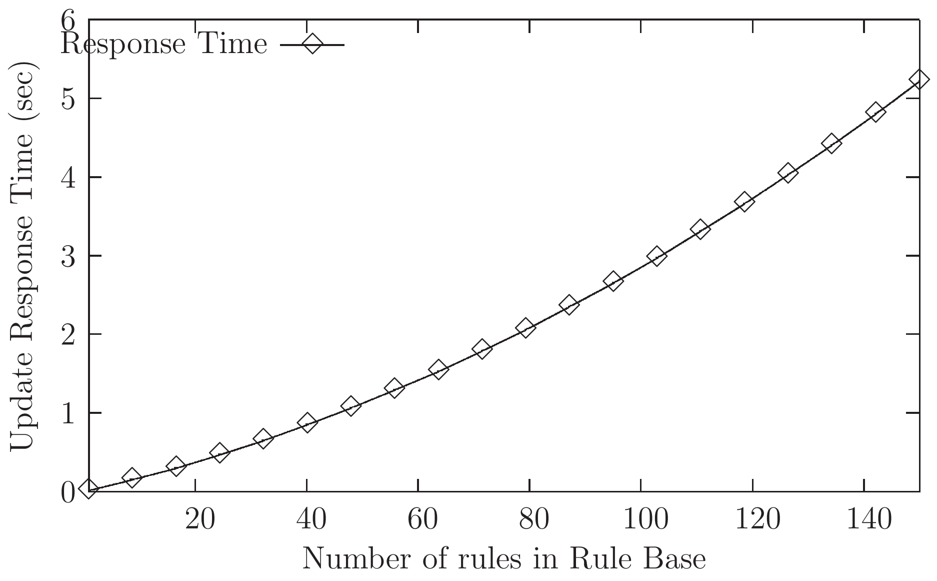


Figure 4.8: Analytical Performance Results — With Index ($p_{mt} = 0.2$)

would be to modify the application code that handles the insertion of a new XML element so that each time a new "`price`" is inserted the current high price for the day is retrieved and, if the new price is higher than this, the current day high is replaced with the new price. The code snippet that would implement this functionality within the application code is described by the following pseudocode:

```
insertElement(element);
if (element.getType() == 'price') then
    dayInfo := element.getAncestorDayInfo();
    high := dayInfo.getDayHigh();
    if (high.getValue() < element.getValue()) then
        high.deleteTextValue();
        high.insertTextValue(element.getTextValue());
    end if;
end if;
```

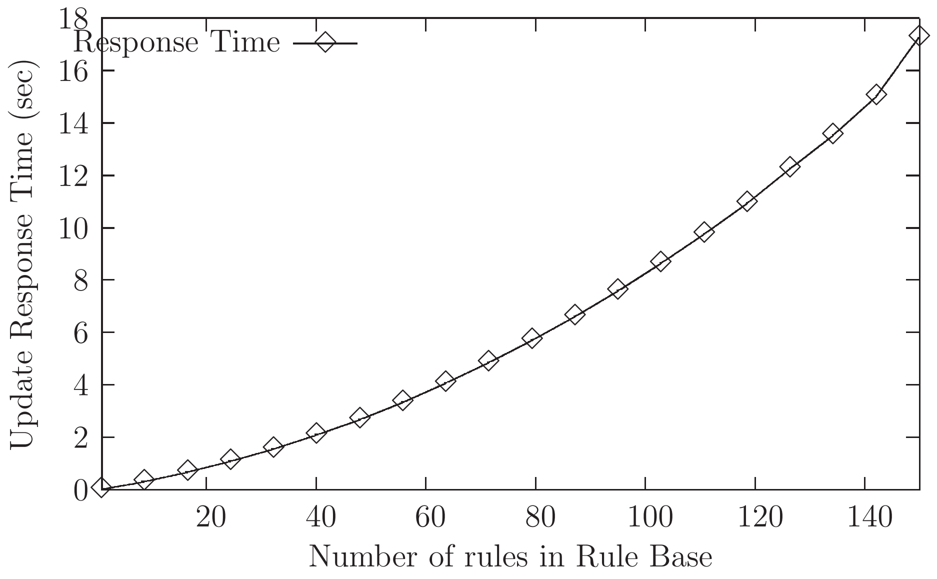


Figure 4.9: Analytical Performance Results — With Index ($p_{mt} = 0.4$)

The corresponding XTL rule is rule $r_1$ given in Section 3.3.1 which checks whether the daily high needs to be updated in response to a new price insertion in some share:

```
on INSERT document('shares.xml')/shares/share/day-info/prices/price
if $delta > $delta/../../high
do DELETE $delta/../../high;
   INSERT <high>$delta/text()</high>
   BELOW $delta/../.. AFTER prices
```

We observe from the above that there is no event detection cost with the hard-coded approach. The cost of condition evaluation is the same in both approaches as they both embed the same condition logic. Similarly, they both execute the same actions and so this cost is also the same in both approaches.

To investigate the update response time for the hard-coded approach, we can therefore modify our analytical model to remove the cost of event detection

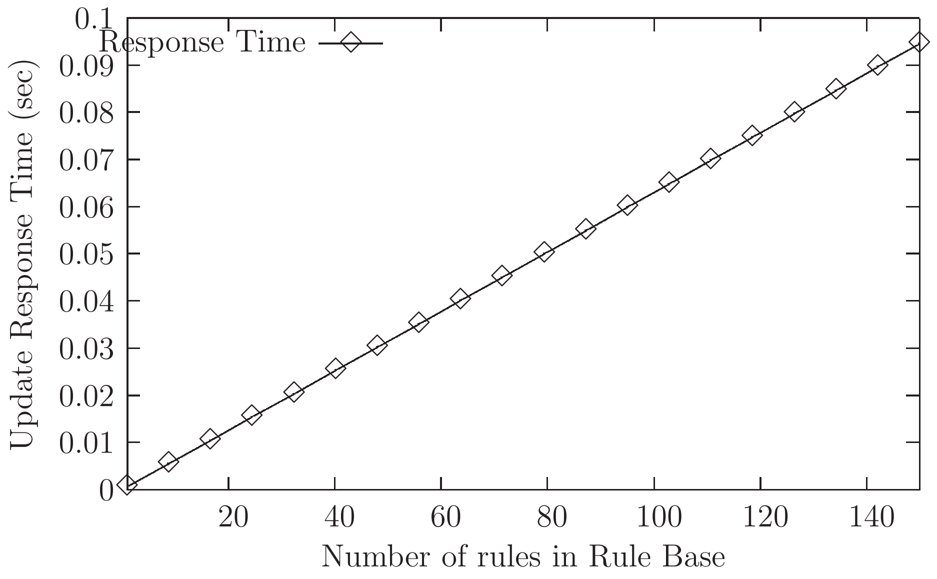


Figure 4.10: Analytical Performance Results — Hard-Coded Case

by setting $\overline{R}^{rule}_{event}(i)$ in equation 4.8 to zero. Figure 4.10 shows how the update response time varies having made this change. We see that the system performs faster and also scales better compared to the ECA rules approach with no rule indexing. Comparing, however, the hard-coded approach with the ECA rules approach when an effective rule index is used (e.g. Figure 4.6) we see that these have similar scalability characteristics.

These performance characteristics of the hard-coded approach make it attractive for applications that require fast response times and have relatively limited or relatively stable reactive functionality. However, for dynamic applications that make extensive use of reactive functionality and require a broad variety of reactive functionalities to be supported, the hard-coded approach may not be attractive. This is because of the higher maintenance cost of changing the code, the possibility of introducing errors in such changes, and the need to redeploy the amended executable. In contrast, with the ECA rules approach, rules can be dynamically added and deleted while the application is running. Furthermore, as shown in

Section 3.5.1, the use of an appropriate rule index can reduce the performance advantages that the hard-coded approach may have over the ECA rules approach.

## 4.6 Summary

In this chapter we have presented a system that implements XTL rules in a centralised environment. We described the system architecture, including the components it comprises, and how XTL rule registration and XTL rule execution are performed. Other key contributions of this chapter include the development of an analytical model for the system, and the presentation of experimental results from an analytical study and from a set of experiments that were conducted on the actual system. An indexing structure for XTL rules was proposed in order to improve the time required to find rules that may be triggered by an event occurrence, and a study of the system performance using this indexing scheme with the analytical model was presented, and these are also contributions of this chapter.

# Chapter 5

# RDFTL: An Event-Condition-Action Language for RDF

## 5.1 Introduction

As discussed in Chapter 1, our involvement in the SeLeNe project [SeLeNe] was the initial motivation to start research on ECA rules for RDF data. ECA rules for RDF could be written on the RDF/XML [Bec04] representation of the RDF data describing learning objects and users, using an ECA language for XML such as XTL described in Chapter 3. Alternatively, rules could be written on the graph representation of the RDF data using a language specifically designed for this data model. We have adopted the second approach since it has a number of advantages compared to the RDF/XML approach:

- It provides a more natural way to handle the RDF graph characteristics — nodes and directed arcs — in their native form, rather than trying to work with a tree-based representation of them.

- The use of an RDF-specific query sublanguage within rules allows us to easily write queries to retrieve parts of the graph, such as arcs, that would be harder to encode in an XML query language on the RDF/XML representation.

- The use of an RDF-specific update sublanguage enables the modification of an RDF graph with respect to the RDF data model and an RDFS schema — e.g. remove an arc or change the classification of an existing resource — that would be harder to express in an XML update language.

- We can exploit the RDF Schema features to give the user a richer set of features for defining rules that conform to RDF Schema restrictions.

- The use of an XML ECA language on the RDF/XML representation would have a performance penalty since the RDF graph would only be traversable in the serialised document order.

This chapter is organised as follows. Section 5.2 gives a description of our ECA language for RDF, called RDFTL, including the syntax and denotational semantics of its query and update sublanguages and the overall syntax of RDFTL rules. Section 5.3 summarises the main points and contributions of the chapter.

## 5.2 The RDFTL Language

RDFTL stands for *RDF Trigger Language*. An early version of it was first described in [PPW03b]. RDFTL is the language supported by our peer-to-peer RDF ECA rule processing system, which we will describe in Chapter 6.

Similarly to XTL, RDFTL also follows SQL3 syntactic approach. RDFTL operates over RDF graphs and complies with the current RDF standards of syntax, semantics and data types [W3C04a, W3C04b, W3C04c]. RDFTL assumes that RDF graphs conform to one or more RDFS schemas. This was mandated by the SeLeNe application domain where every network peer hosts a fragment of a global RDFS schema and each peer's RDF data must conform to this schema. This is in contrast to the XML data warehousing application domain of XTL, which did not have any similar requirement. Another reason for following this approach is because conformance of the RDF data to an RDFS schema has the added advantage of improving the efficiency of searching and indexing the RDF data. An RDF graph conforms to an RDFS schema if the following hold:

(a) every resource in the RDF graph belongs to an RDFS class (in addition to belonging to the default `rdfs:Resource` class);

(b) every property in the RDF graph is declared in the RDFS schema, along with domain and range constraints;

(c) the subject and object of every property in the RDF graph are of the declared subject and object type of the property in the RDFS schema.

When defining an ECA rule in RDFTL, it is necessary to specify the portion of RDF data that each part of the rule concerns: for example, the RDF nodes that will be affected by an event, or the value of an RDF literal used to evaluate a condition. RDFTL uses a path-based query sublanguage for defining queries over the RDF graph, which we discuss next.

### 5.2.1 RDFTL Query Sublanguage

The abstract syntax of the path-based sublanguage is as follows, where $e$ is a query, $c$ is a class name, $p$ is a path expression, $q$ is a qualifier, $uri$ is a URI, a wildcard or empty string, $var$ is a variable, $str$ is a string, $arc\_name$ is a predicate name and $s$ is an RDF literal:

$$
\begin{array}{rcl}
e & ::= & ”resource(”uri”)” \; (”/”p)?[”AS\ INSTANCE\ OF” \; c]|\ var \\
p & ::= & p”/”p \mid p”[”q”]” \mid ”target(”arc\_name”)” \mid ”source(”arc\_name”)” \\
q & ::= & q \; ”and”q \mid q \; ”or” \; q \mid ”not” \; q \mid e \mid e \; ” = ” \; s \mid p” \neq ” \; s|\ e \; “=” \; e \mid e \; “\neq” \; e \\
var & ::= & ”\$” \; str \\
uri & ::= & ” * ”|\ URI \mid \epsilon
\end{array}
$$

The above syntax is similar to that of XPath [W3C99a], except that: $resource(uri)$ matches the resource given by $uri$ in the RDF graph being queried; if $uri$ is an empty string then it matches all the blank resources (with no URI), while if $uri$ is the `'*'` wildcard then it matches all resources regardless of their URI; the optional *AS INSTANCE OF* clause limits the matchings to those that are instances of the specified class; $target(arc\_name)$ returns the set of *object* nodes related by the predicate $arc\_name$ to the set of *subject* nodes given by the context; $source(arc\_name)$ returns the set of *subject* nodes related by the predicate $arc\_name$ to the set of *object* nodes given by the context. The path expressions of RDFTL also resemble those of RDFPath [RDFPath] except that the graph navigation functions in RDFPath are `child` and `parent` instead of `target` and `source`.

For example, the query

`resource(http://www.dcs.bbk.ac.uk/LOs/BK187)/target(dc:type)` expressed in our path sublanguge, applied on the RDF graph shown in Figure 2.4, returns the RDF object obtained by following the arc labelled `dc:type` from the resource with URI `http://www.dcs.bbk.ac.uk/LOs/BK187`.

As another example, the query
`resource()/source(dc:title)[target(dc:type)='Book'] AS INSTANCE OF LO`
returns all instances of the LO class that have a title and are Books.

We give below the denotational semantics of RDFTL's path expressions. We write $S [\![p]\!]$ to indicate the set of nodes selected by path expression $p$ and $S [\![p]\!]\ x$ to indicate the set of nodes selected by path expression $p$ with the node $x$ as context node. We write $Q [\![q]\!]\ x$ to denote whether the qualifier $q$ is satisfied when the context node is $x$[1]. In the denotational specification below, the *value* function returns the value of its argument in the form of a string. The *targets* function takes an RDF predicate $p$ and an RDF subject $x$ as arguments and returns the set $O$ of RDF objects such that, for each $y \in O$, $(x, p, y)$ is a triple in the RDF graph. The argument $p$ may be the wildcard symbol _ in which case $targets(\_, x)$ returns the set of RDF objects $y$ such that $(x, p, y)$ is a triple in the RDF graph for any $p$. Similarly, the *sources* function takes an RDF predicate $p$ and an RDF object $x$ as arguments and returns the set $U$ of RDF subjects such that, for each $y \in U$, $(y, p, x)$ is a triple in the RDF graph. The argument $p$ may be the wildcard symbol _ in which case $sources(\_, x)$ returns the set of RDF subjects $y$ such that $(y, p, x)$ is a triple in the RDF graph for any $p$. After the keyword ON, IF and DO in each part of the rule there may, optionally, be a sequence of *let-expressions* of the form:

`let` *var* `:=` $e$

[1]By convention, when specifying the denotational semantics of a language, arguments of semantic functions which are expressions in the language are delimited by $[\![$ and $]\!]$.

which associate a variable name with a path expression $e$. At run-time, the expression $e$ is evaluated when the processing of the part of the rule in which the `let` expression is defined starts. We assume in the specification of $S, S1, Q$ below, that all variables have been substituted by their corresponding path expression.

$$
\begin{array}{lcl}
S & : & Expression \rightarrow Set(Node) \\
S1 & : & Expression \rightarrow Node \rightarrow Set(Node) \\
S \; [\![resource(uri)]\!] & = & \{y \mid value(y) = uri\} \\
S \; [\![p_1/p_2]\!] & = & \{z \mid y \in S \; [\![p_1]\!], \; z \in S1 \; [\![p_2]\!] \; y\} \\
S \; [\![p[q]]\!] & = & \{y \mid y \in S \; [\![p]\!], \; Q \; [\![q]\!] \; y \; \} \\
S1 \; [\![p_1/p_2]\!] \; x & = & \{z \mid y \in S1 \; [\![p_1]\!] \; x, \; z \in S1 \; [\![p_2]\!] \; y\} \\
S1 \; [\![p[q]]\!] \; x & = & \{y \mid y \in S1 \; [\![p]\!] \; x, \; Q \; [\![q]\!] \; y\} \\
S1 \; [\![target(arc\_name)]\!] \; x & = & \{y \mid y \in \; targets(arc\_name, x)\} \\
S1 \; [\![source(arc\_name)]\!] \; x & = & \{y \mid y \in \; sources(arc\_name, x)\} \\
S \; [\![e \; AS \; INSTANCE \; OF \; c]\!] & = & \{x \mid x \in S[\![e]\!] \wedge \\
 & & \quad c \in targets(\texttt{rdf:type}, x)]\!]\}
\end{array}
$$

$$
\begin{array}{lcl}
Q & : & Qualifier \rightarrow Node \rightarrow Boolean \\
Q \; [\![q_1 \; and \; q_2]\!] \; x & = & Q \; [\![q_1]\!] \; x \wedge Q \; [\![q_2]\!] \; x \\
Q \; [\![q_1 \; or \; q_2]\!] \; x & = & Q \; [\![q_1]\!] \; x \vee Q[\![q_2]\!] \; x \\
Q \; [\![not \; q]\!] \; x & = & \neg(Q \; [\![q]\!] \; x) \\
Q \; [\![p]\!] \; x & = & S1 \; [\![p]\!] \; x \neq \emptyset \\
Q \; [\![p = s]\!] \; x & = & \{y \mid y \in \; S1 \; [\![p]\!] \; x, value(y) = s\} \neq \emptyset \\
Q \; [\![p \neq s]\!] \; x & = & \{y \mid y \in \; S1 \; [\![p]\!] \; x, value(y) \neq s\} \neq \emptyset
\end{array}
$$

The widespread use of path-based query languages, such as XPath, was one reason we chose to adopt a path-based approach for our query sublanguage. In addition, paths seem to be a natural way to navigate graph structures such as RDF and many RDF query languages (RQL, RDQL, SPARQL etc.) support some notion of paths. Furthermore, the ability of the language to retain reasonable expressiveness (and in particular satisfy the SeLeNe requirements) while keeping a simple syntax was an extra motivating factor.

Other RDF query languages might be suitable to express RDF queries in place of our query sublanguage although some further design work might have to be undertaken in order to integrate such languages into our ECA rule syntax. This is because most other RDF query languages are not constrained to just return sets of resources but are also capable of constructing more complex result sets.

### 5.2.2 RDFTL Rule Syntax

Having described the path expressions that RDFTL uses for querying RDF data, we now describe the RDFTL ECA language as a whole, considering in turn the event part, condition part and action part of a rule.

There is an optional preamble to each rule. This preamble may contain one or more clauses of the form `USING NAMESPACE` *name uri* which associate a name with a namespace URI.

Apart from path expressions $e$, described above, the language also employs a syntactic construct for expressing RDF triples. We can have *triples* expressions occurring in the event and the action part of a rule, that are used to indicate modifications to the predicate of a triple, via an insertion, deletion or update. The abstract syntax of *triples* expressions is as follows, where: *triple* is a triple representation; *source_node* is an expression representing the subject of an RDF triple; *target_node*, *old_target_node* and *new_target_node* are expressions representing the object of an RDF triple; *arc_name* is the predicate name of an RDF triple; $s$ is an RDF literal; $e$ is an RDFTL path expression as described above;

*class* is a string representing an RDFS class name; *var* is a variable; and *str* is a string:

$$
\begin{array}{rcl}
triples & ::= & triple \mid upd\_triple \\
triple & ::= & ''('' \; source\_node \; '','' \; arc\_name \; '','' \; target\_node \; '')'' \\
upd\_triple & ::= & ''('' \; source\_node \; '','' \; arc\_name \; '','' \\
 & & \quad old\_target\_node'' \rightarrow ''new\_target\_node \; '')'' \\
source\_node & ::= & e|(\; ''\_'' \; ['' \; AS\ INSTANCE\ OF'' \; class]) \\
arc\_name & ::= & str \mid ''\_'' \\
target\_node & ::= & e|(\; ''\_'' \; ['' \; AS\ INSTANCE\ OF'' \; class]) \mid s \mid var \\
old\_target\_node & ::= & e|(\; ''\_'' \; ['' \; AS\ INSTANCE\ OF'' \; class]) \mid s \mid var \\
new\_target\_node & ::= & e|(\; ''\_'' \; ['' \; AS\ INSTANCE\ OF'' \; class]) \mid s \mid var \\
var & ::= & ''\$''str
\end{array}
$$

The underscore symbol (_) is a wildcard evaluating to all the available subjects, predicates or objects, depending on the place in the triple it is used. Each *triple* expression evaluates to a set of arcs that have as their subject one of the set of resource nodes that the *source_node* expression evaluates to, as their predicate one of the set of predicates that *arc_name* evaluates to, and as their object one of the set of nodes that the *target_node* expression evaluates to. Similarly an *upd_triple* expression evaluates to a set of arcs that have as their subject one of the set of resource nodes that *source_node* evaluates to, as their predicate one of the set of predicates that *arc_name* evaluates to, as their object before they are updated one of the set of nodes that *old_target_node* evaluates to and after they are updated one of the set of nodes that *new_target_node* evaluates to (see below for a discussion of updates of arcs).

The event part of a rule is an expression of one of the following three forms:

1. `[`*`let-expressions`*` IN] (INSERT | DELETE)` $e$

   This detects insertions or deletions of resources specified by the path expression $e$, which evaluates to a set of nodes.

   The rule is triggered if the set of nodes returned by $e$ includes any new node (in the case of an insertion) or any deleted node (in the case of a deletion) that is an instance of *class*, if this is specified[2]. The system-defined variable `$delta` is available for use within the condition and actions parts of the rule, and its set of instantiations is the set of URIs of the new or deleted nodes that have triggered the rule.

2. `[`*`let-expressions`*` IN] (INSERT | DELETE)` ***`triple`***

   This detects insertions or deletions of arcs specified by *triple*. The rule is triggered if an arc labelled *arc_name* from *source_node* to *target_node* is inserted/deleted. The variable `$delta` has as its set of instantiations the arcs which have triggered the rule. The individual components of one these arcs can be identified by `$delta.source`, `$delta.arc_name` or `$delta.target`, containing, respectively, the URI of the resource, the name of the property and the resource URI or the literal value.

3. `[`*`let-expressions`*` IN] UPDATE` ***`upd_triple`***

   This detects updates of arcs specified by *upd_triple*, which has the form *(source_node, arc_name, old_target_node* $\rightarrow$ *new_target_node)*. Here, *old_target_node* is where the arc labelled *arc_name* from *source_node* used to point before the update, and *new_target_node* is where this arc points after the update.

[2]Note that, like XTL, RDFTL supports *semantic* rather than *syntactic* triggering: rule triggering occurs if instances of an event occur and make changes to the RDF graph.

The rule is triggered if an arc labelled *arc_name* from some node *source_node* changes its target from *old_target_node* to *new_target_node*. The variable `$delta` has as its set of instantiations the arcs which have triggered the rule. The individual components of one these arcs can be obtained by `$delta.source`, `$delta.arc_name`, `$delta.old_target` or `$delta.new_target`.

The condition part of a rule may contain an optional *`let-expressions`* `IN` clause followed by a boolean-valued expression. The boolean-valued expression may consist of conjunctions, disjunctions and negations of path expressions, and comparison operators $(=, != , <=, <, >, >=)$ between path expressions and strings. The condition part of the rule may reference the `$delta` variable.

The actions part of a rule is a sequence of one or more actions. Actions can `INSERT` or `DELETE` a resource — specified by its URI — or `INSERT`, `DELETE` or `UPDATE` an arc. Their syntax is as follows for each one of these cases:

1. `[`*`let-expressions`*` IN]`

   `INSERT ‘‘resource(’’ `*`URI`*` ‘‘)’’ AS INSTANCE OF `*`class`*

   `[`*`let-expressions`*` IN] DELETE `*`e`*

   for expressing insertion or deletion of a resource. There is also the syntax:

   *`var`*` := INSERT ‘‘resource()’’ AS INSTANCE OF `*`class`*

   for creating a new blank resource node as an instance of the specified class and assigning this to the specified variable for future reference within the scope of the rule's action part.

2. `[`*`let-expressions`*` IN] (INSERT | DELETE) `*`triple`*` (’,’ `*`triple`*`)*`

   for expressing insertion or deletion of the arcs(s) specified.

3. `[`*`let-expressions`*` IN] UPDATE `*`upd_triple`*` (',' `*`upd_triple`*`)*`

   for updating arc(s) by changing their target node(s).

For insertions, the `AS INSTANCE OF` keyword classifies the new resource to be inserted and is obligatory.

The triples in the case of arc manipulation have the same form as in the event sublanguage. However, the wildcard `'_'` is constrained to appear inside triples in actions as follows. In the case of a new arc insertion, `'_'` is allowed in the place of the *source_node* and has the effect of inserting the new arc for all stored resources. In the case of arc deletion, if `'_'` replaces the *arc_name* then all the arcs from *source_node* pointing to *target_node* will be deleted; if `'_'` replaces the *source_node*, the action deletes all the arcs labelled *arc_name*; replacing the *target_node* by `'_'` deletes the arc *arc_name* from the *source_node* regardless of where it points to. In case of an arc update, `'_'` can be used in place of the *source_node* or the *old_target_node*; in the first case, it indicates replacement of the target node for all arcs labelled *arc_name*; in the second case, use of `'_'` indicates update of the target node regardless of its previous value. The use of combinations of the above wildcards in a triple is also allowed, in order to express more complex update semantics that combine those given above.

We now give two examples of RDFTL rules referring to the RDF Schema in Figure 2.5 showing a fragment of Learning Object metadata and User metadata, as created for the SeLeNe project [SeLeNe].

**Example** Suppose a Learning Object (LO) is inserted whose subject is the same as one of user 128's areas of interest. Then, the following rule $r_1$ adds a new arc linking the newly inserted LO to user 128's personal messages:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
             /target(sl_user:messages) IN
   INSERT ($msgs,sl_user:newLO,$delta);;
```

Here, the event part checks if a new resource belonging to the `LO` class has been inserted. The condition part checks if the inserted LO has a subject which is the same as of one user 128's areas of interest. The `LET` clause defines the variable `$msgs` to be user 128's messages. Finally, the `INSERT` clause inserts a new arc from `$msgs` to the newly inserted LO.

**Example** As another example, if the description of a LO whose subject is the same as one of user 128's areas of interest changes, the following rule $r_2$ inserts a new arc from user 128's `Messages` to the modified LO:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON UPDATE (resource(),dc:description,_->_)
IF $delta.source/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(sl_user:messages) IN
   INSERT ($msgs,sl_user:updated_LO,$delta.source);;
```

### 5.2.3 RDFTL Update Sublanguage

We now specify the denotational semantics of RDFTL's update sublanguage using three functions named *INSERT, DELETE* and *UPDATE*. These accept as input the current database *db* and the update expression (a path expression or a triple), and return the new database that results from the execution of the update. Thus, their type is $Update\_Expr \rightarrow Database \rightarrow Database$. We write: $INSERT\ [\![expr]\!]db$, $DELETE\ [\![expr]\!]db$, or $UPDATE\ [\![expr]\!]db$, to indicate an insertion, a deletion or an update according to the expression *expr* with respect to the database *db*. These functions encompass the constraints that need to be checked and the low level actions that take place during the execution of a specific update. Since the RDF database can be seen as a set of triples, these functions treat all updates as triple insertions, deletions and updates with respect to the input database *db*. In the specification of the functions below, we assume that all variables have been substituted by their corresponding path expression.

For the purposes of the denotational specification below, we use a $createResource(uri)$ function that creates and returns a new resource node with the given $uri$, if specified, and otherwise creates a blank node. In the specifications below $INSERT[\![(se, p, te)]\!]db$ and $DELETE[\![(se, p, te)]\!]db$ correspond to `INSERT` ***`triple`*** and `DELETE triple` actions respectively, where *se* is the expression defining the source of the triple, *p* is the arc and *te* is the expression defining the target of the triple. Note that *INSERT* and *DELETE* each apply to one triple expression only: in case that more than one triple expression is specified in the action syntax, the *INSERT* or *DELETE* function is applied to each of them in turn. $INSERT[\![(resource(uri), c)]\!]db$ and $INSERT[\![(resource(), c)]\!]db$ correspond to

`INSERT ‘‘resource(’’` ***`URI`*** `‘‘)’’ AS INSTANCE OF` ***`class`*** and

`INSERT ‘‘resource(’’ ‘‘)’’ AS INSTANCE OF` ***`class`*** actions respectively, where $uri$ is the URI of the inserted resource and $c$ is the RDFS class the resource belongs to[3]. $DELETE[\![e]\!]db$ and $DELETE[\![(e,c)]\!]db$ correspond to `DELETE` ***e*** and `DELETE` ***e*** `AS INSTANCE OF` ***class*** respectively, where $e$ is the path expression that evaluates to the set of the resources to be deleted and $c$ is the RDFS class to which the resources to be deleted should belong. Finally, $UPDATE[\![(se,pe,te,te')]\!]db$ corresponds to the `UPDATE` ***upd_triple*** action, where again $se$ is the expression defining the source of the triple, $pe$ is the arc, $te$ is the expression defining the old target of the triple and $te'$ the expression defining the new target of the triple. Similarly to triple *INSERT* and *DELETE*, the *UPDATE* function applies to one triple expression only; if more than one triple expression is specified in the action syntax, the *UPDATE* function is applied to each of them.

$$
\begin{aligned}
INSERT[\![(se,p,te)]\!]db \quad = \quad & db \;\cup\; \{(s,p,t) \mid \\
& (if\;\;(se \neq '\_')\;then\;s\;\in\;S[\![se]\!] \\
& else\;s\;\in S[\![resource(*)]\!]) \wedge \\
& t \in S[\![te]\!] \wedge \\
& \Big(\exists\; x\; s.t.\; (p, \texttt{rdfs:domain}, x) \in db \wedge \\
& \quad (s, \texttt{rdf:type}, x) \in db\Big) \wedge \\
& \Big(\exists\; y\; s.t.\; (p, \texttt{rdfs:range}, y) \in db \wedge \\
& \quad (t, \texttt{rdf:type}, y) \in db\Big) \\
& \}
\end{aligned}
$$

[3]The $INSERT$ function does not capture the transient variable assignment associated with the second of these

$$
\begin{array}{lcl}
DELETE[\![(se,p,te)]\!]db & = & db - \{(s,p,t) \mid \\
 & & (if\ (se \neq '\_')\ then\ s\ \in\ S[\![se]\!] \\
 & & else\ s\ \in S[\![resource(*)]\!]) \wedge \\
 & & (if\ (te \neq '\_')\ then\ t\ \in\ S[\![te]\!] \\
 & & else\ t\ \in S[\![resource(*)]\!]) \\
 & & \} \\
INSERT[\![(resource(uri),c)]\!]db & = & if\ \Big(\ \nexists\ (y, \texttt{rdf:type}, z) \in db\ s.t. \\
 & & \quad value(y) = uri \wedge \\
 & & \quad (z, \texttt{rdfs:subClassOf}, \texttt{rdfs:Resource})\Big) \\
 & & \quad then\ db \cup \{(createResource(uri), \texttt{rdf:type}, c)\} \\
 & & else\ db \\
INSERT[\![(resource(),c)]\!]db & = & db \cup \{(createResource(), \texttt{rdf:type}, c)\} \\
DELETE[\![e]\!]db & = & DELETE[\![(e,\_,\_)]\!](DELETE[\![(\_,\_,e)]\!]db) \\
DELETE[\![(e,c)]\!]db & = & DELETE[\![e\ AS\ INSTANCE\ OF\ c]\!]db \\
UPDATE[\![(se,p,te,te')]\!]db & = & INSERT[\![(te,p,te')]\!](DELETE[\![se,p,te]\!]db)
\end{array}
$$

## 5.3 Discussion

To our knowledge, RDFTL is the first ECA language that has been proposed specifically for RDF. This has involved studying the SeLeNe requirements regarding reactive functionality, studying the RDF and RDFS data model, designing the path and update sublanguages to meet the SeLeNe requirements and finally combining them all to form the whole ECA language syntax. In this chapter we have presented the syntax of the language, and the denotational semantics of its query and update sublanguages. In the next chapter we propose an architecture and a prototype implementation to support the processing of RDFTL rules in P2P environments. We also describe the rule execution semantics in the proposed P2P architecture, considering aspects of rule execution such as rule coupling mode, rule prioritisation, set- and instance-oriented rules, and distributed rule execution.

For the future we plan a detailed study that will investigate the query and update expressiveness of RDFTL over relational data, as well as the precise extensions that may be required for it to become query and update complete for relational data.

In particular, a study similar to that for XTL that was presented in Chapter 3, can be conducted for investigating the query capabilities of RDFTL. Given a mapping between relational and RDF schema constructs we will have to show that RDFTL is at least as powerful as the $while_{new}$ language. This requires showing that RDFTL can emulate the constructs of the *while* language, and then that it can also emulate the extensions to *while* forming the $while_{new}$ language. Regarding the update capabilities of RDFTL, we can again use the results presented in [AV91] regarding update languages over relational data, to identify the set of updates that RDFTL can express and investigate the potential set of language extensions that will make it update complete over relational data.

Continuing our discussion of ECA rules over RDF data, in the next chapter we propose an architecture and a prototype implementation to support the processing of RDFTL rules in P2P environments.

# Chapter 6

# RDFTL Rules in P2P Environments

## 6.1 Introduction

Following the description of the syntax and execution semantics of the RDFTL language described in Chapter 5, as a proof of concept we have developed a prototype system that implements RDFTL rules in a P2P environment and we present this system in this chapter. The motivation for this work was the SeLeNe project [SeLeNe], and our RDFTL prototype system was envisaged to provide the reactive functionality of SeLeNe in a P2P environment over RDF data.

The chapter is organised as follows: In Section 6.2 we give an overview of P2P systems, for the purpose of providing general background information for the reader, as well as introducing some key aspects of relevance to RDFTL. In Section 6.3 we describe the architecture of our system and its various components. In Section 6.3.1 we describe how RDFTL rules are registered in our system. Section 6.3.2 discusses rule execution in this P2P environment and illustrates this by an extended example. Section 6.3.3 gives further details of our system's service based architecture, including the services provided by peers and superpeers, the way these services interact with each other, and a discussion of possible concurrency control and recovery mechanisms for our system. Finally, Section 6.3.4 discusses how RDFTL address SeLeNe's requirements regarding reactive functionality.

## 6.2 Overview of P2P Systems

In this section, we give an overview of P2P systems, for the purpose of providing general background for the reader, as well as introducing some key concepts of relevance to RDFTL, namely, structured and schema-based P2P networks, routing indexing techniques that resemble our indexing approach, and key P2P network topologies, some of which are used in our analytical model and simulations in Chapter 7.

Depending on how the peers are linked to each other, P2P networks are classified as *unstructured* or *structured* [CCR04]. An unstructured P2P network is formed when the links between the participating peers are established arbitrarily. Such networks can easily be constructed as a new peer that wants to join the network can copy the existing connection links of another peer and then form its own links over time. In an unstructured P2P network, if a peer wants to find a piece of data in the network, the query has to be flooded through the network in order to find as many peers as possible that share the data. The main disadvantage of this approach is that there is no guarantee that the queries will always be resolved while flooding causes a high amount of traffic in the network, thus resulting is a poor search performance. Most of the popular P2P networks such as Gnutella [Gnutella] and Kazaa [Kazaa] are unstructured.

Structured P2P networks follow a discipline in the way the peers are connected with each other which imposes a network topology or a pattern of peer organisation that can then be exploited for message routing. Examples of structured P2P networks are Chord [SMK$^+$01], Pastry [RD01] and CAN [RFH$^+$01]. If the data that are exchanged between the peers conform to a schema then the P2P network is termed a *schema-based* one. Edutella [NWQ$^+$02], ICS FORTH SQPeer [KC04] and Piazza [TIM$^+$03] are examples of schema-based P2P systems. A peer organisation approach that some schema-based P2P system use is to group the peers into *peergroups* each of which is supervised by a special peer called a *super-peer*. Generally, superpeers have higher computational power compared to simple peers. Each peer is member of at least one peergroup and communication between peergroups is via their superpeers.

Efficient data retrieval is of fundamental concern in P2P networks and data indexing techniques aim to provide this. Data indexing in P2P systems has much in common with indexing in distributed databases. In distributed databases, indexing is provided within the local databases, with a global catalog recording metadata about the fragmentation and allocation of data (the global catalog is typically distributed over the network, rather than centralised or fully replicated [OV99]). Distributed database architectures make two assumptions that are not generally the case in P2P systems: (i) the nodes are stable and connected in a reliable way with the network; and (ii) the number of nodes participating in the system is small and known in advance. These features make the task of indexing in P2P systems more complex and several approaches have been proposed:

Napster uses a centralised index that is fully replicated on a number of servers, while Gnutella uses no indexes at all, instead flooding the network with queries up to a maximum number of nodes called the *horizon*. Freenet [Freenet] creates fully distributed indexes on each peer of the network, employing key-based routing in which a unique key is associated with each file. Indexing schemes based on Distributed Hash Tables (DHT) have also been proposed and are supported by some research P2P systems such as CAN [RFH$^+$01], Chord [SMK$^+$01], Pastry [RD01] and Tapestry [ZHS$^+$04].

Distributed Hash Tables (DHT) [SMK$^+$01] are a distributed version of the hash table data structure. In a file-sharing P2P network, a DHT partitions a set of *keys*, corresponding to the files, among the participating peers in order to be able to efficiently route messages to the unique owner of each key. Each participating peer is analogous to an array slot in a hash table. The main disadvantage of DHTs is that they support only exact-match search.

In [KP03] two multi-level versions of *Bloom filters* [Blo70] are proposed for filtering path queries on XML documents stored in the nodes of a P2P system. Each node maintains two types of filters, a *local filter* containing a summary of locally stored documents, and a *merged filter* summarising the documents of the nodes directly connected with this node. Both types of filters are updated as appropriate in response to XML data updates. Content-based clustering of nodes is also used in order to further improve query routing over the P2P system.

In [GWJD03] a catalogue framework for locating data in P2P systems is proposed, based on DHTs. It uses the Chord P2P protocol for the management of the messages exchanged between peers. In the case of XML documents, element tags and attribute names are used as the keys of the DHTs. To each of the keys there corresponds a data summary of the nodes containing unique paths to this key (element or attribute). Each new node joining the network creates a local catalogue and contacts any other node of the system in order to populate its own data catalogue. When a node leaves the system, it hands over its catalogue information to its connecting nodes.

The concept of a Routing Index (RI) is introduced in [CGM02]. Routing indexes give a "direction" in which to find the document requested rather than its actual location as would be provided by traditional indexes. This use of routes means that the routing indexes depend on the number of neighbours of each node, and thus on the employed network topology. The objective of a routing index is to allow a node to select the "best" neighbours to send a query to. The notion of goodness may vary depending on the application but it should reflect the number of relevant documents in each peer. A routing index assumes that peers maintain a *local index* of their documents so as to be able to find quickly a requested document. A *Compound RI* (CRI) also records at each peer the number of documents stored at each of its neighbours on each topic of interest. Topics of interests are an application-dependent categorisation of the documents according to their contents. Given such a CRI, what needs to be computed is the "goodness" of each peer for a given query. What is used as a measurement of goodness is the possible number of relevant documents stored at or reachable from the neighbour peers, estimated using some application-dependent algorithm. The main limitation of CRI is that it does not take into account the number of "hops" necessary to reach a document in the network. To tackle this limitation, a *hop-count* routing index is introduced in [CGM02], and an *exponential* routing index is proposed as a way to minimise the storage and message transmission needs of the hop-count routing index.

Schema-based routing indexes have been proposed for the indexing of RDF data in the Edutella system [NWS+03]. Edutella uses two kinds of routing indexes: Super-Peer/Peer Routing Indexes (SP/P RI) and Super-Peer/Super-Peer Routing Indexes (SP/SP RI). An SP/P RI stores information about the data usage in each peer of its peer group. This includes information such as the schemas (e.g. `dc` or `lom` that are namespace prefixes of Dublin Core [DC03] and IEEE LOM [IEEE-LOM] metadat schemas, respectively) and properties (e.g.

`dc:subject`) used, as well as possibly conventional indexes on property values. When a peer registers with a superpeer, it provides the superpeer with its data usage, a process called advertisement. The peer undertakes to keep this information up-to-date by informing its superpeer each time that a change affecting the advertised data takes place. At each super-peer, query fragments are matched against the SP/P RIs in order to determine peers that are relevant to this query (although this gives no guarantee that the returned result set from a peer will not be empty). A similar approach is used in SP/SP RIs which are essentially a summary of all the local SP/P indexes. SP/SP indexes contain the same kind of information as SP/P indexes but they associate it with the neighbour superpeers in order to assist routing. Queries are forwarded to neighbour superpeers based on SP/SP indexes and from there to the connected peers based on the SP/P indexes. When a peer registers with a superpeer it sends all its schema information to the superpeer. The superpeer matches this information against its existing SP/P index to check if new elements have to be added in order to include the new peer into the SP/P index. If so, the new elements are added to the index and a message is broadcast to the rest of the superpeers for them to update their SP/SP indexes accordingly.

Data indexing is an effective way to improve query performance in P2P networks, but there are also other factors affecting the performance of P2P systems. Since any routing protocol is dependent on the P2P network's topology, clearly this can significantly affect the query performance of a P2P system. Most of the proposed network topologies are based on optimising graph structures, imposing also some constraints on the link density and link distribution between the peers in order to ensure a minimum level of network robustness. Optimising the topology graph involves minimising the time and the space costs of message transmission. Minimising the time requires a short *graph diameter*[1], which can be achieved by adding links between the network nodes, while minimising the space consumption requires reducing the number of messages sent, which generally requires removing links between the nodes of the network. A network topology also needs to provide some redundancy in order to allow tolerance to node failures.

Proposed topologies include the classic centralised topology in which peers are connected to a central node that manages all the communication between them; the completely decentralised topology proposed by Gnutella in which peers are randomly connected with each other; the ring topology, where the peers are arranged in a ring [SMK$^+$01]; the hierarchical topology, where the nodes are structured in a tree-like hierarchy, e.g. DNS servers; the hypercube topology [SSDN02], where nodes are organised in n-dimensional cubes; and combinations of these.

HyperCuP [SSDN02, NWS$^+$03] can be considered as a hybrid of the hypercube and centralised topologies as superpeers are connected with each other in a hypercube fashion and the rest of the peers are directly connected to superpeers in a client-server fashion. A HyperCup network has a diameter $log_2 N$ where $N$ is the number of nodes; it guarantees that upon a message being broadcast each node receives the message exactly once and that exactly $N-1$ hops are required to reach all nodes in the network. Furthermore, it ensures high tolerance in node failures.

## 6.3 RDFTL P2P System Architecture

We have implemented a system for processing RDFTL rules in P2P environments. The rule processing functionality in our system is provided by a set of services that constitute the *RDFTL ECA Engine*. This set of services acts as an 'active'

[1]The *diameter* of a graph is the greatest distance between any two of its vertices.

wrapper over a distributed set of ‘passive’ RDF/S repositories, exploiting their query, storage and update functionality.

Our system is an example of a schema-based P2P system, and it has an architecture similar to the superpeer-based architecture of Edutella. Similar to Edutella, our system allows hybrid schema and data fragmentation with possible replication between peers. SQPeer [KC04], for example, also does not impose any particular data fragmentation or replication policy; each superpeer integrates a set of peers that support the same schema, although a peer may belong to more than one peergroup if it supports more than one schema; this is in contrast to our approach where each peer is connected to one superpeer only and all peers support the same RDF schema. Piazza [TIM+03] focuses on the semantic integration and global querying of heterogenous data distributed over a P2P network, where each peer supports its own schema.

The architecture of our system is illustrated in Figure 6.1. Each superpeer shown in the figure supervises a group of peers, its peergroup, as well as itself hosting a fragment of the global RDF/S data stored in the network. At each superpeer there is an ECA Engine installed. Each peer or superpeer hosts a fragment of an overall global RDFS schema, and each superpeer’s RDFS schema is a superset of its peergroup’s individual RDFS schemas. In general, superpeers, and indeed peers, may hold heterogeneous RDFS schemas, and there is a need for an RDFS schema mapping service. The techniques discussed in [CKK+03, MP03] could be used as the basis for such a service. This is beyond the scope of this thesis and an interesting area of future extension of our system.

The fragment of the global RDFS schema stored at a peer may change as a result of changes in the peers’ RDF/S data. Peers notify their supervising superpeer of any updates to their local RDF/S repository. Peers may dynamically join or leave the network at any time.

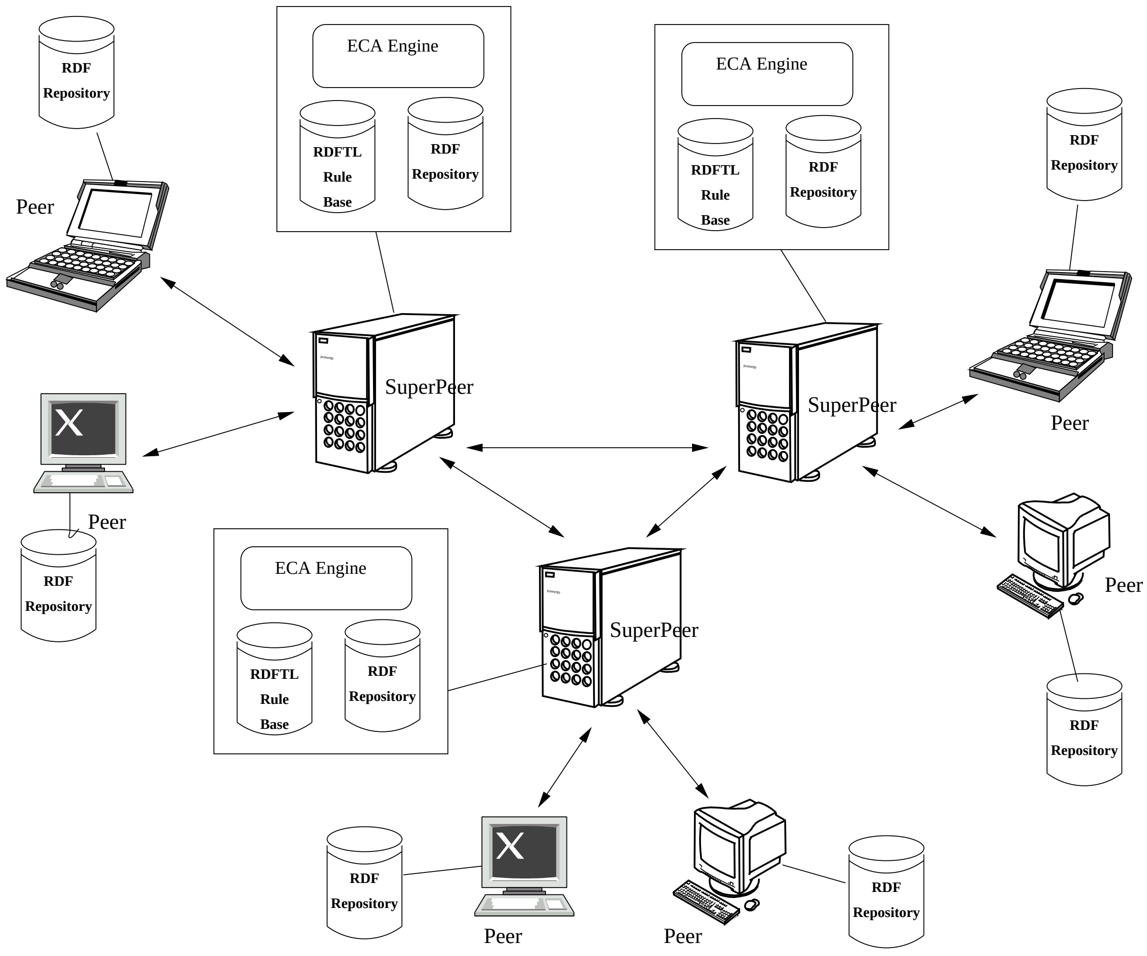


Figure 6.1: RDFTL P2P System Architecture

Each superpeer defines access privileges to other superpeers over the classes and properties in its RDFS schema. These privileges may be read-only, read-write or private, describing the corresponding access level to the instances of each class and property. More fine-grained access privileges are also allowed on specific RDF resources and properties.

In the dynamic applications that we envisage, such as SeLeNe, ECA rules are likely not to be hand-crafted but automatically generated by higher-level presentation and application services. An ECA rule generated at one site of the network might be triggered, evaluated, and executed at different sites. Within the event, condition and action parts of ECA rules there might or might not be references to specific RDF resources, i.e. ECA rules may be *resource-specific* or *generic*.

Whenever a new ECA rule $r$ is generated at a peer P, it will be sent to P's superpeer for storage. From there, $r$ will also be forwarded to all other superpeers, and a replica of it will be stored at those superpeers where an event may occur that may trigger $r$'s event part, i.e. those superpeers that are **e-relevant** to $r$ (see below). A rule $r$ has a globally unique identifier of the form $SP_i.j$, where $SP_i$ is the originating superpeer identifier and $j$ a locally unique identifier for the rule in $SP_i$'s rule base.

We assume that at run-time rules are triggered by updates occurring within a single peer's local RDF repository, i.e. there is no need for distributed event detection. We also assume that each particular copy of a rule's action part executes within a single peer's RDF repository, i.e. there is no need for distributed update execution. If there is a need to distribute a sequence of updates across a number of peers in reaction to some event, then rather than specifying one rule of the form

$$\texttt{on}\ e\ \texttt{if}\ c\ \texttt{do}\ a_1, \ldots, a_n$$

$n$ rules $r_1, \ldots, r_n$ can be specified, where each $r_i$ is of the form `on` $e$ `if` $c$ `do` $a_i$ and where $r_1$ *prec* $r_2$ *prec* $\ldots$ *prec* $r_n$. Note that it is also possible to relax the total ordering of $r_1, \ldots, r_n$ into a partial ordering, or no ordering at all. However, there is still the limitation that at runtime a copy of each $a_i$ will only execute on one peer. This event detection and update execution functionality is sufficient for SeLeNe, though generalising our techniques and architecture to support distributed event detection and distributed update execution are areas of possible future work.

Given an RDF schema $S$ and an RDFTL rule $r$, we now define what it means for $r$ to be **relevant** to $S$. There are three types of relevance:

- $r$ is **e-relevant** to $S$ if each of the path expressions that either appear in the event part of $r$ or are used by the event part through variable references, can be evaluated on $S$. e-relevance is used in our system for determining which rules should be replicated in the rule bases of which superpeers, and this depends on the data stored in that peergroup. If $r$ is e-relevant to the schema, $S$, of a superpeer, it will be stored in the superpeer's rule base. We require that each of the path expressions in the event part of $r$ exists in $S$ because we assume that there is no distributed event detection.

- $r$ is **c-relevant** to $S$ if some step in one of the path expressions referenced by the condition part of $r$ can be evaluated on $S$. c-relevance is an indication of which peers and superpeers may participate in the evaluation of a condition (we assume that conditions may be evaluated at multiple sites).

- $r$ is **a-relevant** to $S$ if all actions in the action part of $r$ are **a-relevant** to $S$. a-relevance is used in our system for determining at which peers instances of the rule action part will be executed (we assume that there is no distributed update execution).

An individual action is a-relevant to $S$ if it satisfies one of the following:

– If it is a deletion or insertion of resources that uses `AS INSTANCE OF` ***`class`***, then ***`class`*** must be in $S$.

– If it is a deletion of resources that does not use `AS INSTANCE OF` ***`class`***, then we determine the most specific class of resources that the path expression in the deletion would return. This class must be in $S$.

– If it is an action over triples that uses a property $p$, then $p$ must be in $S$. If it is a deletion of triples that uses the wildcard '_' instead of a property (the only action allowed to do this), then the classes of resources returned by the path expressions involved in the deletion must exist in $S$. Note that use of the wildcard '_' instead of the source or target node of a triple would return all resources.

**Example**: Consider the following RDFTL rule:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
DO INSERT resource(http://www.dcs.bbk.ac.uk/users/129) AS INSTANCE OF User;
  INSERT ($delta,dc:subject,_);
  DELETE ($delta,dc:reader,_);
```

and the RDF schema given in Figure 2.5. According to the above definitions, the rule is **e-relevant** to the schema as the class `LO` exists in the schema. The condition part is **c-relevant**, as both steps (`$delta` and `target(dc:subject)`) of its path expression can be evaluated on the schema. On the other hand, the action part of the rule is not **a-relevant** to the schema as the second action of the rule is not a-relevant to the schema because the `dc:reader` property is not a part of the schema.

We say that a peer or superpeer is e-relevant, c-relevant or a-relevant to a rule $r$ if $r$ is e-, c- or a-relevant, respectively, to the peer or superpeer's RDFS schema.

### 6.3.1 RDFTL Rule Registration

As already mentioned, whenever a new ECA rule $r$ is registered at a peer P, it is sent to P's supervising superpeer for syntax validation, translation into the local repository's query and update language, and storage. From there, $r$ will also be sent to all other superpeers, and a replica of it will be stored at those superpeers that are e-relevant to $r$.

Determining the e-, c- and a-relevance of a particular ECA rule to a superpeer involves comparing the path expression(s) used by that part of the rule against the superpeer's RDFS schema. In order to aid this comparison, an index can be kept at each superpeer which we discuss further in Section 6.3.3.

Each superpeer matches each part of the new rule against its indexes, and annotates the event, condition and action parts of the rule with the IDs of local peers which may be affected by each part of the rule. Using the *Routing Service* (see Section 6.3.3), a new rule is propagated to all superpeers of the network and it is stored at those superpeers that are e-relevant to it.

Each superpeer is responsible for specifying the precedence relationship between rules generated by itself or its local peergroup. As rules are propagated from superpeer to superpeer, local decisions are made at each superpeer regarding the precedence of the rules originating from other superpeers compared with its own rules, and a local precedence scheme is applied, e.g. timestamp order, assigning higher priority to local rules, assigning higher priority to more specific rules, or combinations thereof (in our current implementation we use timestamp order).

Changes in a superpeer's index (caused by changes in its peergroup's or its own RDF/S metadata) require the annotations of each part of each rule in its rule base to be updated. Any rules that are no longer e-relevant to the superpeer can be deactivated. Conversely, if the RDFS schema of a superpeer SP changes from having no metadata associated with a particular schema node to now having such metadata, this change is notified to SP's neighbouring superpeers. If any of these neighbours have ECA rules which may have been made e-relevant by the new change, they send these ECA rules to SP. These superpeers also request from their neighbours (other than SP) their current set of ECA rules which are potentially e-relevant to the change, and they forward these rules on to SP. This process repeats until all the potentially e-relevant ECA rules throughout the network have been sent to SP.

## 6.3.2 P2P Rule Execution

In our P2P environment, the RDF graph is partitioned amongst the peers and each superpeer manages its own local execution schedule. Each execution schedule at a superpeer is a sequence of updates constituting fragments of global transactions which are to be executed on the fragment of the global RDF graph which is stored at the superpeer or its local peergroup. Each superpeer coordinates the execution of transactions that are initiated by that superpeer, or by any peer in its local peergroup.

Whenever an update $u$ is executed at a peer P, P will notify its supervising superpeer SP and in particular the *Event Handler* service. SP will consult its rule index through the *Rule Base Indexer* to determine all rules that may be triggered by the $u$. We discuss rule indexes further in Section 6.3.3. The rules that may be triggered according to the index, are then checked to see if their event part is annotated with P's ID. If a rule $r$ may have been triggered and its event part is annotated with P's ID, then SP will send $r$'s event query to P to evaluate.

If rule $r$ has indeed been triggered, its condition will need to be evaluated, after generating an instantiation of it for each value of the `$delta` variable if this is present in the condition. The distributed evaluation of the condition is coordinated by SP's *Condition Evaluator* service. Our current implementation assumes that all conditions can be evaluated within the local peergroup and does not support distributed query processing across a number of superpeers. Subqueries of the condition part of a triggered rule are dispatched to the appropriate peers in the peergroup for evaluation by their Query Managers. A superpeer can use the annotations on a rule's condition part to determine to which local peers subqueries of the condition should be dispatched for evaluation. If the `$delta` variable is present in the condition, it will have been instantiated and the superpeers' indexes can be consulted for more precise information about which local peers are relevant to subqueries of the instantiated condition.

If a condition evaluates to true, then the control is passed to the *Action Scheduler* service in SP to proceed to the action part processing. The *Action Scheduler* generates from the action parts of rules that have fired a list of updates to be considered for execution. The Action Scheduler maintains the local execution schedule and sends each instance of a rule's action part (there will be only one instance if the rule is a set-oriented rule, and one or more instances if the rule is an instance-oriented rule) to its local peers, according to the annotations on the rule's action part made during the rule's registration. The instances of the rule's actions part will also be sent, via the *Routing Service*, to all neighbouring superpeers and from there in turn to all other superpeers of the network. All superpeers that are a-relevant to the rule will consult the access privileges to their data in order to decide whether the updates they have received can be scheduled and executed on their local peergroup.

In practice, to detect possibly non-terminating rule executions, a maximum number of recursive rule firings is allowed. In the current version of the system, no transaction management has been implemented. So each time the Action Scheduler dispatches an update for execution somewhere in the network, we assume that this execution is successful. It is planned that future versions of the system will support transaction management as discussed in Section 6.3.5 below.

We now specify the execution model of RDFTL rules in a P2P environment, in the form of pseudocode that shows how rules are processed at one superpeer, `sp_i`. As mentioned above, each superpeer `sp_i` maintains its own execution schedule `s_i`, and each peer and superpeer manages its own RDF data repository. The schedule at a superpeer consists of a list of pairs $(ap_l, delta_l)$, where $ap_l$ is the whole action part of some rule $r_l$ and $delta_l$ is a set of instantiations for the `$delta` variable. The schedule `s_i` which initiates rule execution at a superpeer `sp_i` consists of a single rule action part and set of instantiations for the `$delta` variable.

The event/condition coupling mode is Immediate (as in XTL) but the condition/action coupling mode is Detached coupling (compared to Immediate in XTL). The reason that we have chosen Detached condition/action coupling for RDFTL is because detached transactions have fewer management requirements, simplifying transaction monitoring in the P2P environments which are the focus of the RDFTL language and its implementation[2].

The pseudocode expressing rule execution at a superpeer `sp_i` is as follows,

[2]Using Detached coupling mode may make transaction management in a distributed environment an easier task, but the loose relationship between the parent and the child transactions compared to the other coupling modes may make it harder to enforce stricter transaction control rules that may be necessary in some applications. This is an area of further investigation.

and we explain below its main aspects:

```
while s_i != [] do {
   (ap,delta) := head(s_i);
   s_i := tail(s_i) ;
   ap_peers := getActionPartAnnotations(ap);
   ap_instances := createInstances(ap,delta);
   for each instance a_k in ap_instances do {
       for each peer p_j in ap_peers do {
           db_j := getRepository(p_j);
           for each action a in a_k do {
               (changes_j,db_j) := updateDB(db_j,a);
               for each rule r_l in the rule base of sp_i do {
                   if changes_j[l] != {} then {
                       cpeers := getConditionPartAnnotations(r_l);
                       (value,delta) := evalCondition(l,changes_j[l],cpeers);
                       if value = True then {
                          s_i := s_i ++ [(actionPart[l],delta)];
                          dispatchActionToOtherSPs((actionPart[l],delta));
                       }
                   }
              }
          }
       }
   }
}
```

In the above pseudocode, the function `head` returns the first element of a list and the function `tail` returns the rest of the list.

The function `getActionPartAnnotations` returns the set of peer IDs the action part is annotated with. The function `createInstances` generates the instances of an action part given a `delta`: if the action part does not reference the `$delta` variable, one instance is generated, otherwise the set of instances is generated by substituting occurences of `$delta` within the action part by each member of `delta` (so if $n$ is the cardinality of `delta`, $n$ instances of the action part will be generated).

The `getRepository` function returns the RDF data repository of a peer. The function `updateDB` executes an action `a` on a repository `db_j` returning a pair `(changes_j,db_j)`, where `db_j` is the updated RDF data repository at `peer_j` after the execution of action `a`, and `changes_j` is an array such that `changes_j[l]` is the set of newly inserted or newly deleted nodes corresponding to the event part of each rule `r_l` in the rule base of the current superpeer, `sp_i`.

In particular, if `a` is an insertion then for each `r_l` which may be triggered by `a`, the event part of `r_l` is evaluated on `db_j` after the execution of `a` and `changes_j[l]` is the intersection of this result and the set of new items inserted by `a`; if `a` is a deletion then for each `r_l` which may be triggered by `a`, the event part of `r_l` is evaluated on `db_j` before the execution of `a` and `changes_j[l]` is the intersection of this result and the set of items that are subsequently deleted by `a`.

Note that in the actual system implementation, the `updateDB` function executes asynchronously on peer `p_j`. Here though, for the purposes of brevity and simplicity, we assume a sequential execution of all the functions, omitting the intermediate network access and message passing processes described earlier.

The function `getConditionPartAnnotations` returns the set of peer IDs, `cpeers`, that the condition part of a rule `r_l` is annotated with. The function evalCondition then coordinates the, possibly distributed, evaluation of the rule r_l's condition part on the repositories of cpeers. If the $delta variable does not occur in the condition, then the condition is evaluated just once, the variable value is set to the result and the variable delta is set to changes_j[l]. If the $delta variable occurs in the condition, then the condition is evaluated once for each member of changes_j[l], the subset of changes_j[l] for which it evaluates to True is determined, the variable delta is set to this subset, and the variable value is set to True if delta is non-empty and otherwise is set to False. If value is True, then the action part of rule r_l, actionPart[l], and the delta set are suffixed onto the current schedule s_i, as well as being dispatched to all other superpeers. We now give an example that illustrates both rule registration and rule execution in our system.

**Example** Consider the P2P network shown in Figure 6.2 and assume the peer and superpeer repositories store fragments of the RDF/S schema of Figure 2.5.

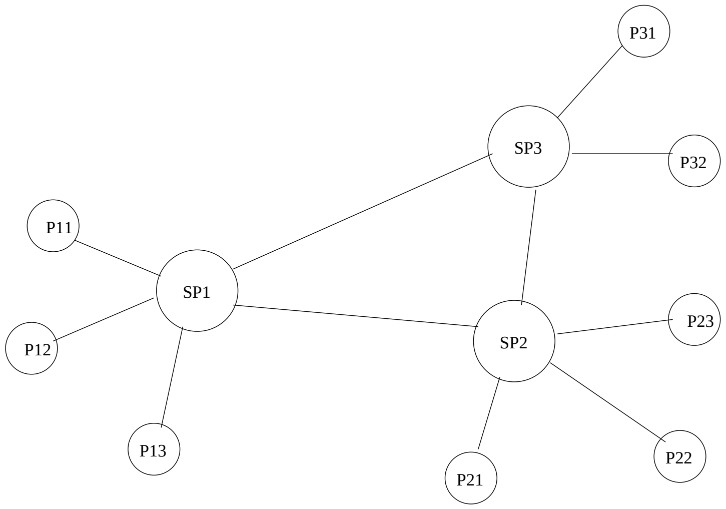


Figure 6.2: Example P2P network

Suppose the rule r1 below is submitted for registration at peer $P_{21}$ of the peergroup of superpeer $SP_2$:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject) =
  resource(http://www.dcs.bbk.ac.uk/users/128)
  /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
               /target(sl_user:messages) IN
INSERT ($msgs,sl_user:newLO,$delta);;
```

This rule will be sent by $P_{21}$ to its supervising superpeer $SP_2$ for registration.

There, the e-, c- and a-relevance of `r1` with the respect to the schemas of $SP_2$ and its peergroup will be determined. Let us assume that, as a result, the event part and condition part of `r1` are annotated with the ID of peer $P_{21}$, and its action part with the ID of peer $P_{22}$. The rule is also checked for its syntactic validity and is then translated into the corresponding query expressions of the underlying RDF repository (for the event and condition part) and the appropriate update-API function calls (for the action part) and is stored in $SP_2$'s rule base.

The rule `r1` is also propagated to all the other superpeers where the same process is repeated. Let us assume this results in the replication of `r1` at $SP_1$, with its event part being annotated with $P_{11}$ and $P_{12}$, its condition part annotated with $P_{12}$ and $P_{13}$ and its action part is annotated with $P_{13}$.

Consider a second rule, `r2` below, also submitted for registration at peer $P_{21}$ and assume that this rule is stored in the rule base of $SP_2$, annotated with $P_{21}$ on its event part and condition part and with $P_{23}$ on its action part, and is also replicated in the rule base of $SP_3$, annotated with $P_{31}$ and $P_{32}$ on its event part and condition part and with $P_{32}$ on its action part.

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT (resource() AS INSTANCE OF LO,dc:description,_)
IF $delta.source/target(dc:subject) =
  resource(http://www.dcs.bbk.ac.uk/users/128)
  /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
               /target(sl_user:messages) IN
INSERT ($msgs,sl_user:updated_LO,$delta.source);;
```

Finally, a third rule `r3` below is submitted for registration at peer $P_{12}$ and is stored in the rule base only of $SP_1$, annotated with $P_{12}$ on its event part and condition part and with $P_{13}$ on its action part.

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:type) = 'Book'
DO INSERT ($delta,dc:description,'Computer Science Book');;
```

The contents of the three rule bases at $SP_1$, $SP_2$, $SP_3$ after these three rules have been registered are as shown in Figure 6.3.

Now consider an update that inserts the metadata shown in Figure 6.4 at peer $P_{12}$. $P_{12}$ notifies $SP_1$ of the insertion events that have occurred. For each insertion, $SP_1$ consults its Rule Base Indexer to determine the rules that may be triggered by the event. This results in the rules `r1` and `r3` being identified as possibly having been triggered. These both have their event part annotated with $P_{12}$, so $SP_1$ sends $P_{12}$ their event queries to evaluate. In both cases, the

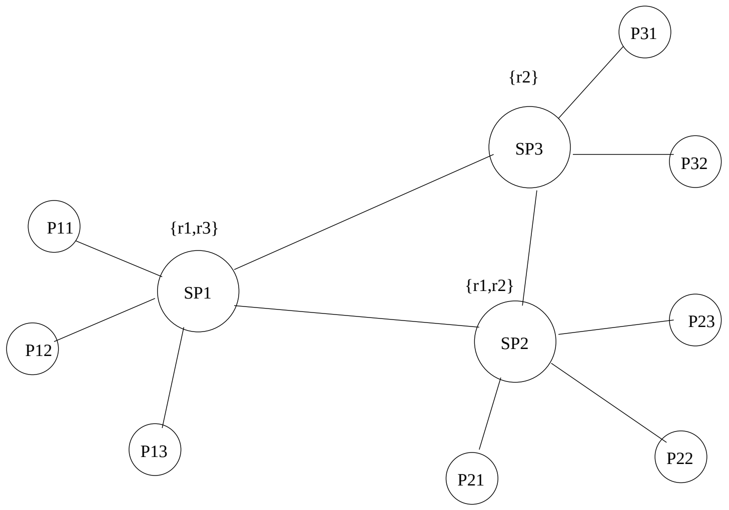


Figure 6.3: Example P2P network with the rules at each superpeer

resulting `delta` set will contain the newly inserted LO and both rules will indeed be triggered.

The Condition Evaluator of $SP_1$ will evaluate the condition part of both of the rules, using the annotations on the condition parts to coordinate their evaluation over its peergroup ($P_{12}$ and $P_{13}$ in the case of `r1` and $P_{12}$ in the case of `r3`). As can be seen from Figure 6.4 and Figure 2.4, the condition of `r1` will evaluate to False while that of `r3` will evaluate to True for the single LO instance contained in the `delta` set.

The Action Scheduler of $SP_1$ then undertakes the execution of the action part of `r3` by placing it on its own execution schedule and also dispatching it to the other superpeers. This will result in the execution of the action part of `r3` at the local peer $P_{13}$, and suppose it also results in its execution at the remote peers $P_{21}$, $P_{31}$ and $P_{32}$ (because that action part is determined to be a-relevant to those peers). Each one of these peers will inform its supervising superpeer, i.e. $SP_1$, $SP_2$ and $SP_3$, respectively, of the insertion event that has occurred there. $SP_1$ will find no rules that may be triggered by this event and so rule processing ends

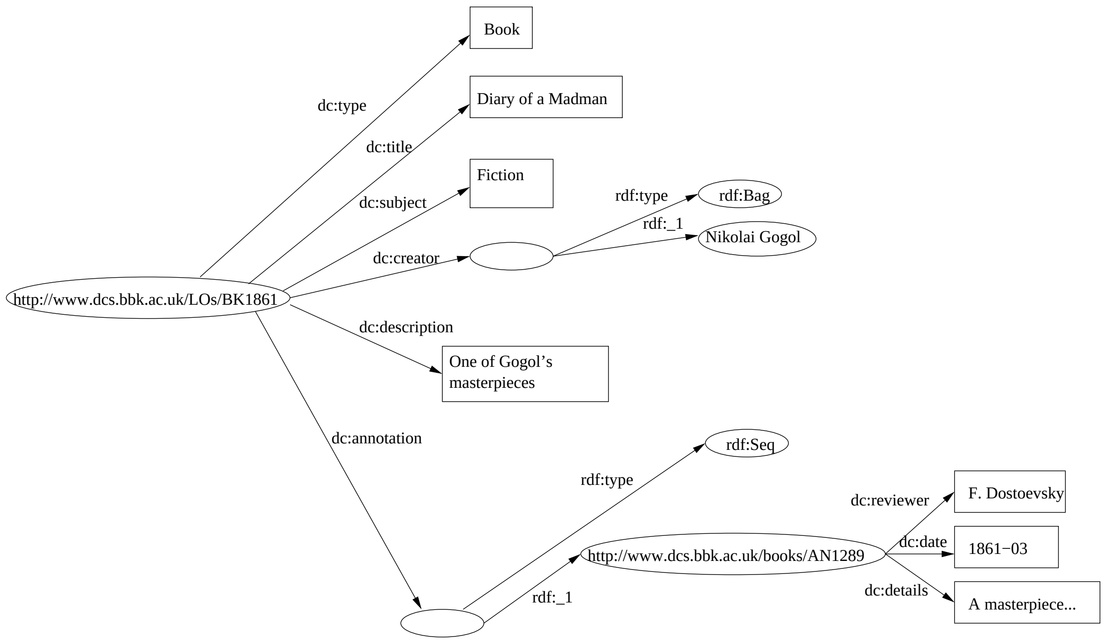


Figure 6.4: Inserted Learning Object

there. $SP_2$ and $SP_3$ will both find that rule `r2` may be triggered, and so rule processing will carry on there (with this small set of example rules, no further rules will be triggered).

We assume that instance-oriented rules are *well-defined* implying that if different instances of an instance-oriented rule's action part are executed by different superpeers, then the orders of execution of these different instances is immaterial and the coordinating superpeer does not have to enforce any particular ordering. Moreover, the resulting subtransactions do not have to be executed in isolation from each other. If instance-oriented rules are not well-defined, then different RDF graphs may result from different orders of execution of a set of instances of a rule's action part, violating the determinism of rule execution.

Similarly, we assume that the rules that have the same precedence *commute* meaning that the coordinating superpeer does not have to enforce any particular order of execution of such rules and the resulting subtransactions do not have to be executed in isolation from each other. If rules with the same precedence do not commute, different order of execution of such rules may result in different RDF graphs, again violating the determinism of rule execution. [PPW06] discusses both of the above assumptions and gives conservative tests for verifying these properties for RDFTL rules.

### 6.3.3 Service-based Architecture

Each peer of our P2P network implements a set of core services that provide the basic functionality necessary to participate in the system: local RDF/S event detection, local RDF/S indexing, RDF repository connectivity, and messaging. In order for a host to become a member of the network, it has to support this set of core services. Superpeers in addition support RDFTL rule registration and rule processing, superpeer RDF/S indexing, and manage connectivity with the peers in their peergroup as well as with their neighbouring superpeers.

Our implementation of these services is flexible, allowing a peer to dynamically extend its set of services and become a superpeer, or a superpeer to dynamically shed its extra services and become a simple peer. Below we describe the major services, and their role in the overall operation of the system. Figure 6.5 gives an overview of how these services are distributed in peers and superpeers.

#### Core Services

**Event Detection Service.** This is responsible for detecting data modification events at a peer. It notifies the Event Handler service at its superpeer (see below) of each event occurrence, including the type of the event, the data affected and the time the event occurred. It also notifies its own Peer Indexing service to update its indexes according to the changes, if necessary.

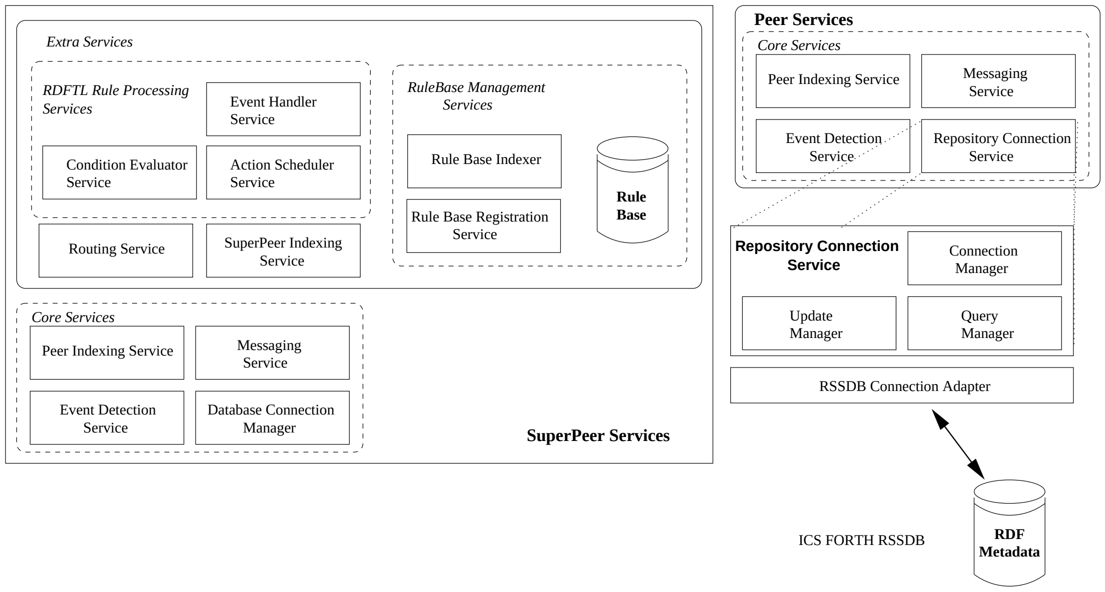


Figure 6.5: Service Distribution in Peers and Superpeers

**Peer Indexing Service.** This maintains indexes on the RDF data stored at the peer and provides a simple query interface over these indexes. After notification of a data change by the Event Detection service, the Peer Indexing service updates, if necessary, the local peer indexes. If necessary, it also sends a notification to the Superpeer Indexing Service (see below) at its supervising superpeer in order for it to update its superpeer indexes.

In more detail, as the RDF data stored at a peer P changes over time, P maintains a list of the RDFS schema classes that have instances in the RDF data stored at P. We call this list the *class index*. This information is also propagated to P's supervising superpeer SP, which maintains a combined version of this information in a table so that each tuple of which contains the name of a class and the set of peers in this peergroup that have data that are instances of this class (a set of peer IDs). Over time, the data stored at P may change so that a class ceases to have any instances in the peer or an RDF resource is inserted that is currently not in P's class index. Such changes are reflected in P's class index and are also propagated to SP.

As well as this class index, each peer also keeps for each of the classes in this list a list of the RDF resources of this type that its RDF data references — we call these lists of RDF resources the *resource index*. Each peer also keeps a list of the properties that its RDF data references — which we call the *property index*. Each superpeer keeps a consolidated resource index and consolidated property index for its entire peergroup.

Our indexing approach is similar to that proposed in [NWS+03] but since we want to maintain more precise information about where various fragments of metadata reside in the network and, as far as possible, do not want unnecessary routing of queries and updates to peers and superpeers that are not relevant, we have adopted the approach of maintaining, apart from the schema, also information about the actual resources stored at each peer.

**Repository Connection Service.** This manages the connection with the underlying RDF repository. It consists of three subcomponents: the **Connection Manager** that provides connection pooling, the **Update Manager** that interprets and passes RDF data update requests to the repository, and the **Query Manager** that establishes communication with the query engine of the repository and retrieves query results. In the current version of our system we are using ICS-FORTH RSSDB [RDFSuite] as the RDF repository. For the future we plan also to support Jena2 [Jena2].

**Messaging Service.** This is responsible for all message exchange between a peer and its supervising superpeer. It undertakes to wrap or unwrap the outgoing or incoming messages, respectively, and pass them to the appropriate service.

### Superpeer Services

**RDFTL Rule Processing Services.** This set of services includes the **Event**

**Handler**, **Condition Evaluator** and **Action Scheduler** services already mentioned earlier. The Event Handler receives notification regarding event occurrences in the supervised peers, and it undertakes to send the event part expressions of the rules that may be triggered to the appropriate peers and then pass those that will actually be triggered to the Condition Evaluator. The Condition Evaluator evaluates the condition part of the triggered rules before it passes those that evaluate to True to the Action Scheduler for the execution of their action part expressions.

**Rule Base Management Services.** This set of services is dedicated to maintaining the local rule base, including indexing of its contents and providing simple query and update functionality over it.

The **Rule Registration Service** receives a new RDFTL rule and undertakes to register it in the rule base. An **RDFTL Language Interpreter** is first invoked to translate RDFTL path expressions to the corresponding query expressions of the underlying RDF repository, and RDFTL rule actions to update-API function calls. The **Superpeer Indexing** service is then contacted in order to construct the list of peers that are affected by each part of the rule, using the indexes at the superpeer. Each part of the rule is annotated with the list of relevant peers and then the annotated rule is stored in the rule base.

The translated, but not yet annotated rule, is also sent to the neighbouring superpeers, using the Routing Service, and from there to all superpeers in the network. It will be stored in the rule base of all superpeers that are e-relevant to it, after it has been appropriately annotated.

The **Rule Base Indexer** creates and maintains an index on the contents of a rule base, aiming to speed up the discovery of rules that may be triggered by an event. Whenever the rule base is updated (i.e. a rule is added or deactivated) this service undertakes to perform the appropriate updates to the rule indexes.

Our rule index operates as follows:

The event part of an RDFTL rule can be split into the event type and the part of the RDF graph that is affected by the event, i.e. resource or arc.

An index entry is created for each rule registered at the superpeer. The index entry is a tuple that has different contents depending on whether the rule refers to a *resource-based* or an *arc-based* event. The contents of the tuple in the case of a resource-based event are:

- the name of the rule;
- the type of the event (INSERT or DELETE); and
- the class that the resource is an instance of; if no class is specified, the default is `rdfs:Resource`.

while the contents of the tuple in the case of an arc-based event are:

- the name of the rule;
- the type of the event (INSERT, DELETE or UPDATE);
- the name of the arc modified, or a '`*`' if a wildcard is specified;
- if specified in the rules
  - the class that the source node of the arc is an instance of;
  - the class that the target node of the arc is an instance of; and
  - the class that the new target node of the arc is an instance of, if the type of the event is UPDATE.

- Again if no class is specified in any of the above, then the default is `rdfs:Resource`.

Due to the tuple-based form of the index, the two types of tuples can be seen as two relational tables and can be indexed and queried as such. This approach becomes even more convenient due to the fact that most RDF repositories and programming frameworks, including Jena2 and RDFSuite, are based on relational database systems to store and query the RDF data, thus eliminating the need to set up a separate database to use for our rule index. For the resource-based index there is a hash index on the type of event and class attributes, while for the arc-based index there is a hash index on the type of event, the arc name and all the class attributes.

When the superpeer receives a notification of an update $u$, the update is examined to determine: (a) if it modifies an RDF resource or an arc, (b) the type of the event and (c) the class of the resource affected, or the name of the arc and the class names of its source and the target nodes[3]. A query is then submitted to the appropriate index table in order to obtain the rules that may be triggered by $u$.

**Example** As an example consider the rules $r_1$ and $r_2$ given in Section 5.2.2. The event part of $r_1$ considers the insertions of a resource that is an instance of class LO. So the resource-based index tuple for $r_1$ will be: `(r_1, INSERT, LO)`. The event part of $r_2$ has an arc-based event that detects the update of the `dc:description` property. So the arc-based index tuple will be: `(r_2, UPDATE, dc:description, rdfs:Resource, rdfs:Resource, rdfs:Resource)`.

Now, assume that the following update occurs:

```
INSERT resource('http://www.dcs.bbk.ac.uk/LO/421')
AS INSTANCE OF LO
```

The following query is submitted to the resource-based rule index, represented

[3]In order to find the class of an RDF resource with URI we issue a query to the RDF/S data of the form `resource(uri)/target(rdf:type)`

by the `resource_based_ix` relational table, and will return the set of rules that may be triggered by the update:

```
SELECT rule_name
FROM resource_based_ix
WHERE action_type = 'INSERT' AND resource_class = 'LO';
```

**Routing Service.** This keeps a list of the neighbouring peers and superpeers in order to maintain the message transmission paths in the network. This service is called each time that another service of a superpeer needs to transmit a message to one or more of its neighbouring peers or superpeers.

**Superpeer Indexing Service.** This is responsible for the creation and maintenance of the consolidated indexes operating at the peergroup level, including the class, the resource and the property indexes. Each time a change occurs to a peer's RDF data, this service is notified, if necessary, by the corresponding Peer Indexing service in order to update these peergroup-level indexes.

### 6.3.4 RDFTL Support for SeLeNe Requirements

P2P was one of the deployment alternatives for a Self e-Learning Network proposed in the SeLeNe project [SC04], the others being 'centralised' and 'mediation-based'. The architecture of our prototype RDFTL rule processing system is consistent with the requirements for SeLeNe's P2P deployment scenario: our system has superpeers and peers organised into peergroups supervised by superpeers; each peer and superpeer hosts a fragment of the overall RDF/S descriptions, and may dynamically join or leave the network. We believe that our system could readily be modified to support the other two deployment alternatives proposed in [SC04]: in the 'centralised' case, a fixed set of central servers would each host the superpeer services while a fixed set of clients would each host the peer services;

in the ‘mediation-based’ case, a set of fixed servers (‘authority sites’) would again each host the superpeer services while a dynamic set of ’providers’ would each host the peer services.

We listed in Section 1.2 of Chapter 1 the aspects of the SeLeNe user requirements that were to be provided by SeLeNe’s reactive functionality. Referring to the RDF data and schema shown in Figures 2.4 and 2.5, we illustrate below how RDFTL rules can be used to achieve the four types of reactive functionality requirements listed in Section 1.2:

(i) *automatic notification to users of the registration of new LOs of interest to them*

According to the SeLeNe Use Case definitions [SC04], when a user submits a request for this kind of notification, the system generates an appropriate ECA rule based on the user’s input. Each time the rule fires, its actions part will add the necessary new information within the “notifications” area of the user’s profile: referring to Figure 2.5, this is represented by the `Messages` class and its properties. SeLeNe’s Presentation Service will check this part of the user profile when a user logs on to the system, and also periodically while they are logged on. Users will be able to view their notification information and to choose whether or not to delete it from their profile.

The first example RDFTL rule given in Section 5.2.2 illustrates the ECA rule aspect of this functionality:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
```

```
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(sl_user:messages) IN
   INSERT ($msgs,sl_user:newLO,$delta);;
```

Here, the event part checks if a new resource that is a Learning Object (i.e. belongs to the `LO` class) has been inserted. The condition part checks if the inserted LO has a subject which is the same as one of user 128's areas of interest. The `LET` clause defines the variable `$msgs` to be user 128's messages. Finally, the `INSERT` clause inserts an arc labelled `newLO` from `$msgs` to the newly inserted LO.

(ii) *automatic notification to users of the registration of new users who have interests in common with them in their personal profile*

This functionality is achieved similarly to (i) and its ECA rule aspect is illustrated by the following rule, which places information about the registration of new users into user 128's collection of messages:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF User
IF $delta/target(sl_user:interest)/target(sl_user:interest_typename)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
    /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)/
                target(sl_user:messages) IN
INSERT ($msgs,sl_user:new_users,$delta);;
```

Here, the event part checks if a new user has been inserted. The condition part checks if the new user has an interest that is the same as one of user 128's interests. The `LET` clause defines the variable `$msgs` to be user 128's messages. Finally, the `INSERT` clause inserts an arc labelled `new_users` from `$msgs` to the newly inserted user's URI.

(iii) *automatic notification to users of changes in the description of resources of interest to them*

This functionality is achieved similarly, and its ECA rule aspect is illustrated by rule $r_2$ given in Section 5.2.2, which places information into user 128's collection of messages about learning objects whose descriptions have been updated and whose subject is the same as one of user 128's interests:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON UPDATE (resource(),dc:description,_->_)
IF $delta.source/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(sl_user:messages) IN
   INSERT ($msgs,sl_user:updated_LO,$delta.source);;
```

(iv) *automatic propagation of changes in the description of one resource to the descriptions of other, related resources, e.g. propagating changes in the description of a LO to the description of any composite LOs defined in terms of it*

Deliverable 4.1 of SeLeNe [RS03] specified an algorithm for inferring the metadata description of a composite LO from the description of its constituent LOs. The algorithm relies on the availability of a taxonomy of terms assumed to be common throughout the SeLeNe system, together with a subsumption relation *contains* between pairs of terms. Users annotate their new LOs with a selection of terms from the taxonomy — by definition, such a set of terms is *reduced* i.e. it does not contain any pair of terms $t$, $t'$ such that $t$ *contains* $t'$. When a composite LO is registered with the system, its description is automatically created from the descriptions of its immediate consituent objects by the algorithm given in [RS03] and the resulting description also has the *reduced* property. Figure 6.6 illustrates a composite LO with URL `O`, its constituent LOs with URLs `O1`, `O2`, ... `On` and the descriptions of each of these LOs, represented by a property `term` linking the LO to a term of the SeLeNe LO taxonomy.

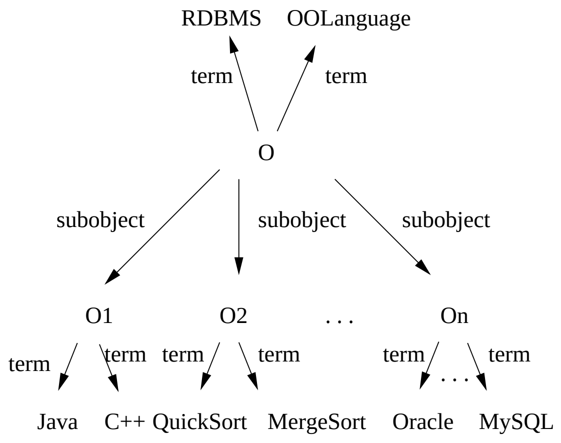


Figure 6.6: Example of Learning Object Descriptions

Part of the reactive functionality envisaged for SeLeNe was to be able to automatically update the description, $T(O)$, of a composite LO, $O$, if a change occurs in the description of one of its constituent LOs, $O_1, \ldots, O_n$. It is possible to incrementally update the description $T(O)$ if a new term $t$

is added to some $T(O_i)$ and retain the *reduced* property:

(a) $t$ should be added to $T(O)$ if it is not related by *contains* to any other member of $T(O)$;

(b) if there is a $t'$ in $T(O)$ such that $t'$ *contains* $t$, then $T(O)$ should not be altered;

(c) if there is a $t'$ in $T(O)$ such that $t$ *contains* $t'$, then $t'$ should be deleted from $T(O)$, $t$ should be added to $T(O)$, and any other $t''$ in $T(O)$ such that $t$ *contains* $t''$ should also be deleted from $T(O)$.

This is achieved by the two RDFTL rules below which respectively implement (a) and (c) (there is no rule needed to cover case (b)). The rules assume that the taxonomy is "complete", i.e. all containment relationships are included explicitly in it as arcs labelled `contains`:

```
ON INSERT (_ AS INSTANCE OF LO, term , _ AS INSTANCE OF Term)
IF not ($delta.target,contains,$delta.source
                /source(subobject)/target(term))
   and not ($delta.source/source(subobject)
                /target(term),contains,$delta.target)
DO INSERT ($delta.source/source(subobject), term, $delta.target);;

ON INSERT (_ AS INSTANCE OF LO, term , _ AS INSTANCE OF Term)
IF ($delta.target,contains,$delta.source
              /source(subobject)/target(term))
DO LET $old_term =
        $delta.source/source(subobject)
           /target(term)[source(contains)=$delta.target] IN
```

```
    DELETE ($delta.source/source(subobject), term, $old_term);
    INSERT ($delta.source/source(subobject),term,$delta.target);;
```

Conversely, if a term $t$ is deleted from $T(O_i)$ then $t$ should be deleted from $T(O)$, if present, provided there is no $t'$ in the description of any of the sub-objects of $O$ such that $t$ *contains* $t'$. This is achieved by this RDFTL rule:

```
ON DELETE (_ AS INSTANCE OF LO, term , _ AS INSTANCE OF Term)
IF ($delta.source/source(subobject),term,$delta.target)
    and not ($delta.target, contains,
                $delta.source/source(subobject)
                 /target(subobject)/target(term))
DO DELETE ($delta.source/source(subobject),term,$delta.target);
```

where LO is an RDFS class representing the learning objects and Term is a class representing the terms within the terms taxonomy.

These rules for incrementally updating the descriptions of composite learning objects would recursively fire each other, all the way up from “base” LOs, i.e. ones that have no constituent LOs, up to “root” LOs, i.e. ones that are not constituent LOs of any other LO. The termination of this rule execution is guaranteed provided that there are no cyclic sub-object relationships, an assumption that was also made in [RS03].

### 6.3.5 Concurrency Control and Recovery

Our current implementation does not yet support any concurrency control or recovery mechanisms, but we briefly discuss here possible implementations of such mechanisms.

In theory, any distributed concurrency control protocol could be adapted to a P2P environment. For example, the AMOR system adopts optimistic concurrency control [HSS03]. The serialisation graph is distributed amongst those peers responsible for transaction coordination, which are analogous to our superpeers. The AMOR system assumes that conflicts are only possible between those transactions that are accessing a particular 'region' of resources (analogous to our peers) and thus subgraphs of the global serialisation graph are stored and replicated amongst those coordinators which service a particular region. However, the regions are not static and these subgraphs are dynamically merged and replicated as transactions execute and regions evolve. We could use similar techniques to merge and replicate subgraphs of the global serialisation graph between our superpeers.

In the classical approach to distributed transactions, global transactions hold on to the resources necessary to achieve their ACID (Atomicity, Consistency, Isolation and Durability) properties until such time as the whole transaction commits or aborts. In a P2P environment this may not be feasible: the resources available at peers may be limited, peers may not wish to cooperate in the execution of global transactions, and peers may disconnect at any time from the network, including during the execution of a global transaction in which they are participating[4]. It is therefore necessary to relax the Atomicity and Isolation properties of transactions.

In particular, child transactions executing at different peers may be allowed to commit or abort independently of their parent transaction committing or aborting, and parent transactions may be able to commit even if some of their child transactions have failed. Child transactions that have committed ahead of their

[4]The cascaded triggering and execution of ECA rules could cause longer-running transactions which may further exacerbate these problems, which is why we have adopted Detached coupling mode for RDFTL.

parent transaction committing can be reversed, if necessary, by executing *compensating* transactions [GMS87, KLS90]. These are generated as transactions execute and they reverse the effects of a transaction by compensating each of the transaction's updates in reverse order of their execution. Generating compensating updates is straight-forward for RDFTL updates: the insertion of a resource or triple is reversed by deletion of the resource or triple, the deletion of a resource or triple by an insertion, and an update by the restoration of the original value. If transactions have read from committed transactions which are subsequently reversed, then a cascade of compensations will result.

Suppose that, as the default, a parent transaction and its immediate child transactions are able to commit independently of each other. An explicit **abort dependency** now needs to be specified for each rule. Using the categorisation given in [RPS95], the possible abort dependencies are as follows, with $T_o$ being the parent transaction and $T_r$ the child child transaction:

- **ParentChild**: If $T_o$ aborts then $T_r$ is to abort.
- **ChildParent**: If $T_r$ aborts then $T_o$ is to abort.
- **Mutual**: If either $T_o$ or $T_r$ aborts then so must the other.
- **Independent**: There is no abort dependency between $T_o$ and $T_r$.

For coordinating the execution of compensating transactions, an abort graph can be maintained that describes the abort dependencies between parent and child transactions.

The abort graph will be distributed amongst the superpeers that participate in the execution of a top-level transaction. The graph will be constructed dynamically as each new child transaction is initiated. In particular, each time a transaction $T_n$ at a superpeer $SP_i$ initiates a new child transaction $T_m$ to be executed at a superpeer $SP_j$ (where it may be that $i = j$) then depending on the abort dependency between $T_n$ and $T_m$, the following actions are taken:

1. **ParentChild**: The identifier of $T_m$ and the superpeer $SP_j$ that it will execute on are transmitted to $SP_i$ and recorded there, together with an arc $T_n \rightarrow T_m$ in the local abort graph at $SP_i$.

2. **ChildParent**: The identifier of $T_n$ and the superpeer $SP_i$ that it is executing on are transmitted to $SP_j$ and recorded there, together with an arc $T_m \rightarrow T_n$ in the local abort graph at $SP_j$.

3. **Mutual**: A combination of the actions for **ParentChild** and **ChildParent** above is taken.

4. **Independent**: No local abort graph is updated.

Using the above, in case of a transaction failure all the necessary information is available in order to initiate a compensating transaction, at any level of nesting of the transaction execution tree.

Figure 6.7 gives an example of a distributed abort graph. In this figure, a failure in transaction $T_7$ at $SP_3$ leads to compensation of $T_7$ at $SP_3$, while a failure in transaction $T_6$ at $SP_5$ initiates a compensating transaction for $T_6$ at $SP_5$, a compensating transaction for $T_1$ at $SP_2$ (due to the **Mutual** abort dependency between $T_6$ and $T_1$), and a compensating transaction for $T_5$ at $SP_2$ (due to the **ParentChild** dependency between $T_1$ and $T_5$).

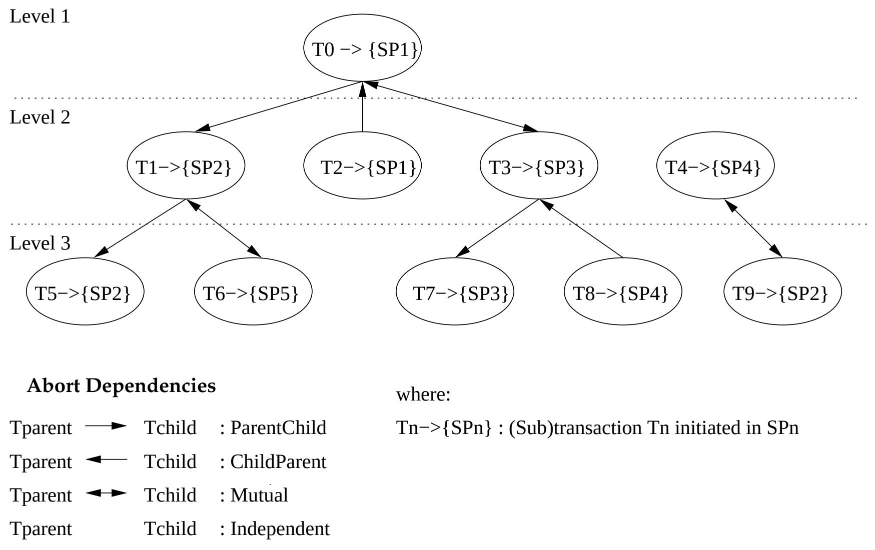


Figure 6.7: Example Abort Graph

## 6.4 Summary

In this chapter we have presented a system that implements RDFTL rules in a P2P environment. We described the system architecture, including the components and the set of services it comprises as well as details of how RDFTL rule registration and RDFTL rule execution are performed in the system. We proposed a scheme for indexing the RDF data stored in repositories of the peers and superpeers, and a scheme for indexing the RDFTL rules stored within the rule base at each superpeer. We examined expressiveness of RDFTL according to how it meets the SeLeNe requirements regarding reactive functionality. Finally, we discussed possible concurrency control and recovery mechanisms for our system.

In the next chapter we conduct a performance study of the system presented here, using analytical and simulation methods, and we present and analyse experimental results aiming to identify the set of factors that the performance of such a system depends on, and obtain insight into the system's scalability.

# Chapter 7

# RDFTL System Performance

## 7.1 Introduction

In this chapter we study the performance and scalability aspects of processing RDFTL rules on RDF data in P2P environments, with a view to determining the practical usefulness of RDFTL for real P2P applications. We develop an analytical model for the system presented in Chapter 6, taking as main performance criterion the average time needed to complete all rule execution resulting from a single update submitted to the system by an application. We present experimental results of an analytical study which examines how this average time varies with the network topology, the number of peers and the degree of RDF data replication between peers. A simulation of the system has also been developed and this is then described, as are the results of similar experiments conducted with this simulation.

The chapter is organised as follows: Section 7.2 develops an analytical model for the system presented in Chapter 6. Section 7.3 presents the experimental results from our analytical study. Section 7.3.4 describes a simulation of the system and presents analogous experimental results obtained using this simulation.

## 7.2 Analytical Model

To our knowledge, ours is the first analytical model for a P2P ECA rule processing system. A performance study of distributed and replicated databases, based on analytical methods, is presented in [NJ00]. A set of alternatives for modelling network communication, data replication and query processing is proposed, and the various interdependencies between the components of the model are discussed. The distributed data model with random data replication resembles the distributed P2P environment we are concerned with in this thesis with the difference that in distributed database systems the nodes are constant and their number is fixed and known, something that is not the case in P2P environments where the number of the peers and the connections between them change dynamically. A description of replication strategies that improve the performance of requests in unstructured P2P networks is presented in [CS02], along with a set of experimental results this is contrast to our structured, schema-based P2P network.

Similar to the XTL rule processing system performance evaluation study, we have chosen *update response time* as the main performance criterion of our RDFTL rule processing system. Other choices could have been system resource consumption (such as CPU usage, memory consumption, number of I/O operations) and network usage, and a detailed examination of these is an area of future work. Other aspects of the system could also be potentially evaluated, such as the cost of updating the data indexes throughout the system and the cost of keeping the rule bases up-to-date.

We define *update response time* to be the mean time taken to complete all rule execution resulting from a single update submitted by a top-level transaction. Transactions consisting of queries and updates are submitted by applications to peers or superpeers. Updates may cause rules to fire, which may in turn cause the firing of further rules, increasing the network traffic as well as the load on peers and superpeers. Queries submitted by a top-level transaction cannot cause rule firing and can only increase the load on peers and superpeers.

In designing the analytical model of the RDTFL processing system, we have made a number of simplifying assumptions as detailed below.

### 7.2.1 Homogeneity Assumption

In common with other performance studies of distributed and replicated databases (e.g. [NJ00]), we make a homogeneity assumption separately for the peers and superpeers of the network. For both these two sets of servers, this assumption asserts identical workloads, the same service capacity and the same amount of data per site. For the superpeers, it also implies the same number of rules per rule base.

For network communication, the homogeneity assumption implies symmetrical communication between superpeers, and between peers and superpeers i.e. the average number of messages from a site $A$ to a site $B$ equals the average number of messages from site $B$ to site $A$. The size of the messages exchanged is considered to be fixed.

We also assume a balanced peer distribution amongst superpeers, with each peergroup having the same number of peers.

### 7.2.2 Rule triggering assumptions

The set assumptions considering the triggering of RDFTL rules is identical to those employed for XTL rules and described in Section 4.4.1. The triggering probabilities and their describing equations remain the same.

$p_{fire}(i)$ denotes, again, the probability that an update occurring at level $i$ of rule execution causes a given rule to fire. This probability depends on the probability $p_{mt}$ that a given rule may be triggered by a given update, the probability $p_t(i)$ that a rule that may be triggered is actually triggered at level $i$, and the probability $p_f$ that a rule that has been triggered actually fires.

$p_{fire}(i)$ is given by the equation:

$$p_{fire}(i) = p_{mt} \cdot p_t(i) \cdot p_f \tag{7.1}$$

$p_t(i)$ is given by the equation:

$$p_t(i) = p_t \cdot p_{reduct}^i \tag{7.2}$$

where $p_t$ denotes the probability that a rule, that may be triggered, is actually triggered as a result of an update.

As a consequence of equations 7.1 and 7.2:

$$p_{fire}(i) = p_{mt} \cdot p_t \cdot p_f \cdot p_{reduct}^i \tag{7.3}$$

implying that $p_{fire}$ also reduces geometrically with the level $i$.

The number of rules that may be triggered by an update (at any level $i$) is given by

$$r_{mt} = p_{mt} \cdot n_{rules} \tag{7.4}$$

where $p_{mt}$ is the probability that a given rule to may be triggered by a given update and $n_{rules}$ is the number of rules in the rule base. The number of rules that are actually triggered at level $i$ is thus given by the following equation:

$$r_t(i) = p_t(i) \cdot r_{mt} = p_{mt} \cdot p_t \cdot p_{reduct}^i \cdot n_{rules} \tag{7.5}$$

Similarly, the number of rules $r_{fire}(i)$ that fire for each event that occurs at level $i$ is given by

$$\begin{aligned} r_{fire}(i) &= p_{fire}(i) \cdot n_{rules} \\ &= p_{mt} \cdot p_t(i) \cdot p_f \cdot n_{rules} = p_{mt} \cdot p_t \cdot p^i_{reduct} \cdot p_f \cdot n_{rules} \end{aligned} \tag{7.6}$$

### 7.2.3 System Modelling

Two kinds of queues are, again, assumed in our model. A *transaction queue* at each peer that accepts queries or updates arising from rule execution, and an *action scheduler queue* at each superpeer that queues transactions resulting from rule firing which are then dispatched for execution to the appropriate transaction queues of peers in the peergroup. For each queue we assume a FCFS (First Come First Served) service discipline. New queries/updates that arrive at a peer are placed at the tail of the peer's transaction queue for execution, and the same happens at the action scheduler queue in superpeers. The choice of FCFS for the action scheduler queue is due to the Detached coupling mode adopted for the RDFTL rule processing system (rather than the Immediate rule coupling mode in the XTL rule processing system).

The query/update arrival rate is modelled as a Poisson process, meaning that the arrival of a new item does not depend on any previous item and the inter-arrival time is exponentially distributed. The exponential distribution of inter-arrival time leads to the service time, for a query to be evaluated or an update to be executed also follow an exponential distribution.

The transaction queues and the action scheduler queues are queues of the type $M/M/1$, where $M$ indicates exponential distribution for both the process arrival rate and the service time, and 1 specifies that each peer or superpeer provides one single service point.

Regarding the network communication within the system, we assume that the communication channel between superpeers, and between superpeers and peers, can be modelled as a server with an infinite number of service points, introducing a delay to all messages passed depending on their size and the network bandwidth. The queue model description of this is $M/D/\infty$, where $D$ indicates deterministic service time and $\infty$ an unlimited capacity network[1].

We assume that the size of messages is fixed at $size_m$ (in bytes), and that the communication bandwidth between superpeers is twenty times greater than the bandwidth between peers and superpeers. So the network communication delay introduced for messages between a peer and a superpeer, $t_{delay}^{P-SP}$, is given by the following (where multiplying by 8 converts the message size $size_m$ from bytes to bits and $bps$ represents the network communication bandwidth in bits per second between a peer and superpeer):

$$t_{delay}^{P-SP} = \frac{8 \cdot size_m}{bps} \tag{7.7}$$

and for messages transmitted between two superpeers the delay, $t_{delay}^{SP-SP}$, is given by:

$$t_{delay}^{SP-SP} = \frac{t_{delay}^{P-SP}}{20} \tag{7.8}$$

### 7.2.4 Modelling Update Response Time

We recall that the update response time is the mean time taken to complete all rule execution resulting from a top-level update submitted to a peer or a

[1]A more accurate but more complex way to model the inter-superpeer communication and the communication between superpeers and peers would be to assume that each communication channel is a queuing network of $N$ $M/M/1$ queues. In this way, we could define that there are $N_{sp}$ channels for the inter-superpeer communication case and $N_p$ channels for the superpeer-peer communication case.

superpeer. This update response time, $\overline{R}_{update}$, can be decomposed as follows:

$$\overline{R}_{update} = \overline{R}_{event} + \overline{R}_{cond} + \overline{R}_{action} \tag{7.9}$$

where $\overline{R}_{event}$ is the mean time taken for all event processing, $\overline{R}_{cond}$ the mean time taken for all condition processing and $\overline{R}_{action}$ the mean time taken for all action processing during the rule execution following a top-level update. We now consider each of these three components in turn.

**Event Response Time ($\overline{R}_{event}$)**

For each level of triggering $i$, the mean time taken to process the event part of all the rules at this level $i$, denoted by $\overline{R}_{event}(i)$, is the sum of the mean time spent in peer database processing at level $i$ and the mean time spent in network transmission at level $i$. The time spent in accessing the Rule Base index is considered negligible compared to the rest and is thus omitted:

$$\overline{R}_{event}(i) = r_{fire}(i-1) \cdot N_{action} \cdot (\overline{T}^{db}_{event} + \overline{T}^{net}_{event}) \tag{7.10}$$

Here, the factor $r_{fire}(i-1) \cdot N_{action}$ represents the total number of individual updates caused by the previous level of triggering, assuming that the action part of each rule contributes $N_{action}$ updates to an action schedule (see below). $\overline{T}^{net}_{event}$ is the mean time spent in network processing for processing the event queries following an event occurrence at some peer. $\overline{T}^{db}_{event}$ is the mean time spent in database processing for processing the event queries following an event occurrence at some peer.

When an event occurs at a peer P, details such as the type of the event, the data items that were affected (i.e. the sets of resources inserted or deleted, or the sets of triples that have been inserted, deleted or updated), and the time the event occurred, are wrapped into a message and transmitted to the coordinating superpeer SP. This determines the set of rules in its rule base that may be triggered by the event. The event part of each of the $r_{mt} = p_{mt} \cdot n_{rules}$ rules that may be triggered is sent to P for evaluation. The evaluation results for each event part are transmitted back to SP where they are matched against the set of data items affected by the event. If the intersection of the two sets is non-empty, then the rule is actually triggered.

These processing steps involve contacting the network services three times, the repository services at peer P $r_{mt}$ times, plus matching of the affected data items against the evaluation results of the event part of each of the $r_{mt}$ rules. The network processing time needed for the event part is therefore

$$\begin{aligned} \overline{T}^{net}_{event} &= t^{P-SP}_{delay} \cdot \frac{m-1}{m} \\ &+ 2 \cdot t^{P-SP}_{delay} \cdot \frac{m-1}{m} \cdot r_{mt} \end{aligned} \quad (7.11)$$

where the factor $t^{P-SP}_{delay} \cdot \frac{m-1}{m}$ represents the time taken for the transmission of the event notification message from a peer P to its supervising SP. $t^{P-SP}_{delay}$ is the constant network delay caused by each message sent, $m$ is the number of peers in a peergroup (including the superpeer itself) and $r_{mt}$ the number of rules that may be triggered. The factor $\frac{m-1}{m}$ represents the probability the event has not occurred on the superpeer, assuming events are equally likely to occur at any peer in the peergroup. If the event does occur on the superpeer, then no network transmission will be necessary.

The total time required, following an event occurrence at P, for processing the event queries at P is:

$$\overline{T}^{db}_{event} = r_{mt} \cdot t_{peer\_total} + r_{mt} \cdot t_q \quad (7.12)$$

Here, $t_{peer\_total}$ is the sum of the time $\overline{W}_{peer}$ spent on waiting in P transaction queue plus the mean time $t_q$ needed for a rule's event query to be evaluated on P repository, so:

$$\begin{aligned} t_{peer\_total} &= t_q + \overline{W}_{peer} \\ &= t_q + \frac{\lambda_{peer} \cdot t_q^2}{1 - \lambda_{peer} \cdot t_q} \end{aligned} \tag{7.13}$$

where $\lambda_{peer}$ represents the transaction arrival rate in peers and is fixed to an average throughout this model.

We have also used the quantity $t_q$ for the time needed at P's superpeer for evaluating the intersection of the results of an event query with the set of data items affected by the event, and again there are $r_{mt}$ such intersections to be evaluated.

Substituting equations 7.11 and 7.12 into equation 7.10, and summing over all $k$ triggering levels, we obtain the mean time taken to process all event queries over all $k$ triggering levels during the rule execution following a top-level update as:

$$\begin{aligned} \overline{R}_{event} = \sum_{i=1}^{k} \Big\{ r_{fire}(i-1) \cdot N_{action} \cdot \Big[ t_{delay}^{P-SP} \cdot \frac{m-1}{m} \\ + r_{mt} \cdot (2 \cdot t_{delay}^{P-SP} \cdot \frac{m-1}{m} + t_{peer\_total} + t_q) \Big] \Big\} \end{aligned}$$

**Condition Response Time ($\overline{R}_{cond}$)**

From the set of may-be-triggered rules, only a subset may actually be triggered. The number of rules $r_t(i)$ that are actually triggered at level $i$ is given in equation 7.5. For each of these $r_t(i)$ rules, the time needed for its condition part to be evaluated is given by the following equation:

$$\overline{R}^{rule}_{cond}(i) = \overline{T}^{db}_{cond}(i) + \overline{T}^{net}_{cond}(i) \tag{7.14}$$

where $\overline{T}^{db}_{cond}(i)$ is the time spent on database processing and $\overline{T}^{net}_{cond}(i)$ the time spent on network transmission.

The condition part of each of the rules triggered needs to be evaluated in a distributed manner, in order to determine the subset of rules that will finally fire. This involves generating the appropriate number of instances of the condition part (one or more in the case of an instance-oriented rule, one in the case of a set-oriented rule), determining to which of the peers in the peergroup subqueries of the condition should be dispatched, transmission of the sub-queries to the appropriate peers, evaluation of the subquery at each peer, and finally transmission of the evaluation results back to the superpeer.

We note here that the current RDFTL implementation assumes that rule conditions are evaluated locally within a peergroup. Global rule condition evaluation across multiple superpeers would need to use global P2P query processing techniques, which would generally increase the update response time of ECA rule processing. Its effect on the observed trends of our experimental results presented in Section 7.3 would clearly depend on the complexity and scalability of the query processing algorithms employed. For example, [ST04] discusses query routing in P2P networks that use the HyperCup topology and shows that schema-based clustering of peers at the superpeers improves the network performance significantly. In order not to confuse the performance of global P2P query processing with the additional ECA rule processing functionality that is the focus of this thesis, we retain in our analytical model the assumption that rule conditions are evaluated locally within a peergroup.

We assume that each condition part generates on average $N_{cond}$ instances. Each of these $N_{cond}$ instances has an average number of $n_{steps}$ steps within its constituent queries. The probability $p_{annot_{cond}}$ that a condition instance is relevant to one of the $m$ peers of the peergroup is expressed by the probability that at least one of the steps within its constituent queries can be evaluated at the peer. Each query "step" is either a specific resource URI or the name of a property. The indexes maintained at each superpeer state which resources are present in the RDF repository of each peer of its peergroup, and also which RDFS properties appear within triples in each of the peers' RDF repositories; we assume here that the cost of index look-up is negligible compared with the other system costs. Thus, $p_{annot_{cond}}$ is given by:

$$p_{annot_{cond}} = 1 - (1 - p_s)^{n_{steps}}$$

where $p_s$ is the probability that a query step can be evaluated at the given peer.

We note that the probability $p_s$ also captures the amount of schema and data replication between peers in the overall network. A higher $p_s$ value implies greater replication and a lower value implies less replication. A value of $p_s = 1$ corresponds to full schema and data replication at all peers in the network.

From the above, the mean time needed for a rule's condition part to be evaluated is:

$$\overline{T}^{db}_{cond}(i) = t_{peer\_total} \cdot N_{cond} \cdot n_{steps} \cdot p_{annot_{cond}} \cdot m \tag{7.15}$$

where the time taken to evaluate one step of a query within a rule's condition part in a peer is $t_{peer\_total}$.

The mean time spent on network transmission for the evaluation of a rule's condition part is given by

$$\overline{T}^{net}_{cond}(i) = 2 \cdot t^{P-SP}_{delay} \cdot N_{cond} \cdot n_{steps} \cdot p_{annot_{cond}} \cdot (m-1) \tag{7.16}$$

with the factor $m-1$ representing the number of peers that each message is sent.

From equations 7.14, 7.15 and 7.16 we obtain the mean time needed for the evaluation of the condition part of a single rule as:

$$\overline{R}^{rule}_{cond}(i) = N_{cond} \cdot n_{steps} \cdot p_{annot_{cond}} \cdot [t_{peer\_total} \cdot m + 2 \cdot t^{P-SP}_{delay} \cdot (m-1)] \tag{7.17}$$

So, the mean time taken to process all condition queries over all $k$ triggering levels during the rule execution following a top-level update is as follows, recalling that $r_t(i) = p_t(i) \cdot r_{mt}$ rules are triggered at the $i$th triggering level:

$$\begin{aligned} \overline{R}_{cond} &= \sum_{i=0}^{k} \left( r_t(i) \cdot \overline{R}^{rule}_{cond}(i) \right) \\ &= r_{mt} \cdot N_{cond} \cdot n_{steps} \cdot p_{annot_{cond}} \\ &\quad \cdot [t_{peer\_total} \cdot m + 2 \cdot t^{P-SP}_{delay} \cdot (m-1)] \cdot p_t \cdot \sum_{i=0}^{k} p^i_{reduct} \\ &= r_{mt} \cdot N_{cond} \cdot n_{steps} \cdot p_{annot_{cond}} \\ &\quad \cdot [t_{peer\_total} \cdot m + 2 \cdot t^{P-SP}_{delay} \cdot (m-1)] \cdot p_t \cdot \frac{1-p^{k+1}_{reduct}}{1-p_{reduct}} \end{aligned} \tag{7.18}$$

**Action Response Time ($\overline{R}_{action}$)**

The number $r_{fire}(i)$ of rules that fire at the $i$th level of triggering is expressed by equation 7.6. The mean time needed for the execution of the action part of each of the rules that fire is again expressed as the sum of the time spent on network transmission and the time consumed on path expression evaluation and update execution at the peers' RDF repositories:

$$\overline{R}^{rule}_{action}(i) = \overline{T}^{db}_{action}(i) + \overline{T}^{net}_{action}(i) \tag{7.19}$$

Each instance of the action part of each rule that fires (one instance in the case of a set-oriented rule, one or more instances in the case of instance-oriented rules) is sent to the peers of the local peergroup according to the annotations on the rule's action part and, after consulting the superpeer resource indexes, also according to the presence of specific URIs at peers. Each instance of the action part of each rule is also sent to all other superpeers of the network. Depending on the data access rights on the schema and resources present at each remote superpeer, the rule's action part instances may be scheduled and executed there.

The time spent in network transmission, according to the above process, is given by the following:

$$\begin{aligned}\overline{T}^{net}_{action}(i) \quad = \quad & \Big[t^{P-SP}_{delay} \cdot p_{annot_{action}} \cdot (m-1) \\ + \quad & \Big(t^{SP-SP}_{delay} \cdot n_{hops} + t^{P-SP}_{delay} \cdot (n-1) \cdot p_{allow} \cdot \\ & p_{annot_{action}} \cdot (m-1)\Big)\Big] \cdot N_{action}\end{aligned} \tag{7.20}$$

Here $n_{hops}$ expresses the average number of hops, i.e. message transmission steps, required for an instance of a rule's action part to be transmitted to all of the superpeers of the network. This number depends on the network topology and the routing strategy followed. At each of the $n-1$ remote superpeers, the instance of the action part will be scheduled for execution depending on the probability $p_{allow}$ that the superpeer has within its peergroup the schema and data the resources necessary to execute the instance, and that its data access privileges allow access to these resources. If the execution is possible, the updates comprising the instance of the action part will be sent to $p_{annot_{action}} \cdot (m-1)$ peers of the peergroup, excluding the superpeer itself. Here $p_{annot_{action}}$ expresses the probability that all the steps within the path expressions of the instance of the action part can be executed at a given peer, and is given by

$$p_{annot_{action}} = p_s^{n_{steps}} \tag{7.21}$$

where, as with rule conditions, $p_s$ expresses the probability that a step can be evaluated at the peer and we assume the same number of steps, $n_{steps}$, in a rule's action part as in its condition part.

As mentioned earlier, instances of rules' action parts are placed on superpeers' action scheduling queues. Recall that a number of instances of an action may be generated if the action contains the `$delta` variable and precisely one instance otherwise. We assume that there are an average of $\ell_{inst}$ instances contributed by an action, and that each rule's action part contains on average $size_{ap}$ individual actions. So the number of updates, $N_{action}$, placed by a rule to the action scheduling queue is:

$$N_{action} = \ell_{inst} \cdot size_{ap}$$

After an instance of the action part is placed on an action scheduling queue at level $i$, each update within it waits for a time of $\overline{W}(i)$ before being dispatched, as part of the whole instance, to the appropriate peers of the local peergroup, where it receives a service time of $t_{srv}$. Thus, the mean time spent on the execution of an update within an instance of an action part within a peergroup is:

$$T_{update\_exec}(i) \quad = \quad t_{srv} + \overline{W}(i) \tag{7.22}$$

where $t_{srv}$ is given by

$$t_{srv} = p_{annot_{action}} \cdot m \cdot t_q$$

since an update will be executed, as part of the instance, on $p_{annot_{action}} \cdot m$ peers of the peergroup and we assume that the time needed for the execution of an update is $t_q$.

The mean waiting time in the $M/M/1$ action scheduling queue at level $i$ of triggering is given by the following equation [Jai91]:

$$\overline{W}(i) = \frac{\lambda_{total}(i) \cdot t_{srv}^2}{1 - \lambda_{total}(i) \cdot t_{srv}} \tag{7.23}$$

Here, $\lambda_{total}(i)$ is the total arrival rate of updates at a peergroup at level $i$ of triggering, given by

$$\lambda_{total}(i) = \lambda_{ext} + \lambda_{int}(i) \tag{7.24}$$

where $\lambda_{ext}$ is the arrival rate of top-level updates that are submitted from outside the rule processing system, and $\lambda_{int}(i)$ is the arrival rate of updates that are generated as a result of rule firing at level $i$.

Each of the externally arriving top-level updates will cause a first level of triggering. Each of the updates contained within the transactions generated as a result of this rule triggering may cause further rules to fire, causing more transaction traffic to be generated and so on. So the total arrival rate of updates at a peergroup at level $i$ of triggering can be expressed as:

$$\begin{aligned}\lambda_{total}(i) &= \lambda_{ext} + \lambda_{ext} \cdot r_{fire}(1) \cdot N_{action} \\ &+ (\lambda_{ext} \cdot r_{fire}(1)) \cdot r_{fire}(2) \cdot N_{action}^2 \\ &+ \ldots \\ &+ (\lambda_{ext} \cdot r_{fire}(1) \ldots \cdot r_{fire}(i-1)) \cdot r_{fire}(i) \cdot N_{action}^i \\ &= \lambda_{ext} \cdot [1 + \sum_{j=1}^{i} N_{action}^j \cdot \prod_{l=1}^{j} r_{fire}(l)] \qquad (7.25)\end{aligned}$$

where $r_{fire}(l)$ is the number of rules that fire at triggering level $l$, and $N_{action} = \ell_{inst} \cdot size_{ap}$ is the total number of individual updates each of the $r_{fire}(i)$ rules generates.

According to equations 7.6 and 7.25, the transaction arrival rate at level $i$ is therefore

$$\begin{aligned}\lambda_{total}(i) &= \lambda_{ext} \cdot (1 + \\ &\qquad n_{rules} \cdot p_{mt} \cdot p_t \cdot p_f \cdot \sum_{j=1}^{i} N_{action}^j \cdot \prod_{l=1}^{j} p_{reduct}^l)\end{aligned}$$

The execution time for all the updates within instances of the action part of a rule at level $i$ is $T_{action\_exec}(i) = N_{action} \cdot T_{update\_exec}(i)$.

The time needed for all the $\ell_{inst}$ instances of a rule's action part to be executed with probability $p_{allow}$ on the rest of the $n-1$ remote peergroups is also $T_{action\_exec}(i)$, and so the total time needed for the execution of the action part of a rule at level $i$ is:

$$\overline{T}_{action}^{db}(i) = T_{action\_exec}(i) \cdot [1 + (n-1) \cdot p_{allow}] \qquad (7.26)$$

Using equation 7.19, the fact that $r_{fire}(i)$ rules fire at level $i$, and that there are $k$ triggering levels, we obtain the time taken for all rule action processing as

$$\begin{aligned} \overline{R}_{action} &= \sum_{i=1}^{k} r_{fire}(i) \cdot \left(\overline{R}^{rule}_{action}(i)\right) \\ &= \sum_{i=1}^{k} r_{fire}(i) \cdot (\overline{T}^{db}_{action}(i) + \overline{T}^{net}_{action}(i)) \qquad (7.27) \end{aligned}$$

into which the right-hand sides of equations 7.20 and 7.26 can be substituted.

## 7.3 Experimental Results

As well as developing the analytical performance model described in the previous section, we have also developed a simulator of the RDFTL system in order to increase the accuracy of the performance study and validate the predictions of the analytical model. We have conducted experiments with the analytical model and with the simulator for two different network topologies, random flooding and HyperCup. We have examined the update response time on each, varying the $n$, $n_{rules}$, $p_s$ and $p_{reduct}$ parameters. We note here that the network topology relates to the communication between superpeers, as each (non-superpeer) peer has only one active network connection with its supervising superpeer. The results of our experiments are discussed in Section 7.3.1 for the analytical model and Section 7.3.4 for the simulator.

For the purposes of these experiments, we have used the actual RDFTL implementation in order to measure $size_m$ and $t_q$, by running a set of sample queries and updates, and the values obtained for these are shown in Table 1. In the absence of information about real RDFTL rule sets for real applications, we have fixed the values of the ECA-rule related parameters $k$, $N_{cond}$, $N_{action}$, $size_{ap}$, $p_{reduct}$,

$p_{allow}$, $p_t$, $p_f$, $p_{mt}$, $\lambda_{ext}$, $bps$ and $n_{steps}$ as shown in Table 1. For the e-learning applications originally envisaged for RDFTL, we believe that these values are representative of the likely rule sets that will arise. The value of $N_{action}$ is relatively low as RDFTL rules operate on metadata and so "bulk" updates are likely to be infrequent. The value of $k$ is of the same order of magnitude as adopted in commercial active DBMS as an upper limit for the number of recursive rule firings allowed. The number of peers in each peergroup is fixed at $m = 20$.

In our performance study below, we consider the general trends of the performance graphs as the primary result rather than the absolute values obtained. We will see that varying the values of the $n$, $n_{rules}$ and $p_s$ parameters affects the absolute performance values but not their general trend.

| **Parameter** | **Base setting** |
|---|---|
| $\lambda_{ext}$ | 20 trans/s |
| $\lambda_{peer}$ | 5 trans/s |
| $size_m$ | 10 kbytes |
| $bps$ | 512 kbits/s |
| $t_q$ | 1 s |
| $k$ | 30 |
| $N_{cond}$ | 4 |
| $N_{action}$ | 8 |
| $size_{ap}$ | 4 |
| $p_{reduct}$ | 0.2 |
| $p_{allow}$ | 0.9 |
| $p_t$ | 0.5 |
| $p_f$ | 0.5 |
| $p_{mt}$ | 0.2 |
| $n_{steps}$ | 4 |
| $m$ | 20 |

Table 7.1: Parameter Base Values

### 7.3.1 Analytical Study Results

In our analysis, we examine for each of the two network topologies the update response time with respect to varying numbers of peergroups in the network ($n$), varying numbers of rules per rule base ($n_{rules}$) and varying data replication ($p_s$).

In the first topology, the superpeers are connected at random and any message between them is broadcast from the originating superpeer to all its neighbouring superpeers, and from there to all their neighbours, and so forth, flooding the network until the message reaches all the superpeers. This simple topology does not place any guaranteed upper bound on the number of hops that will be necessary for a message to reach all superpeers. It also does not prevent a message being received more than once by the same superpeer.

Figure 7.1 shows how the update response time varies with this random topology as the number of peergroups $n$ increases, with $p_s = 0.1$ corresponding to data replication of 10% and the number of rules per rule base $n_{rules}$ set to 100, 500 and 1000. From the shape of these curves it is clear that the system does not scale well.

As the number of peergroups increases, the update response time rapidly rises towards very high values. For a relatively small number of rules per rule base (100) and for up to 10 peergroups we obtain acceptable update response times, due to the relatively low number of available communication paths between superpeers. As the number of peergroups $n$ increases, and with that the number of possible communication paths between superpeers, the rapidly rising values of the update response time make the system unstable and unusable. Similar behaviour is observed with larger values of $n_{rules}$ except that the system becomes unstable at lower values of $n$. Similar sets of experiments conducted for higher values of $p_s$ result in graphs with similar upwards trends, except that the absolute values of the update response time are larger and the system becomes unstable at lower

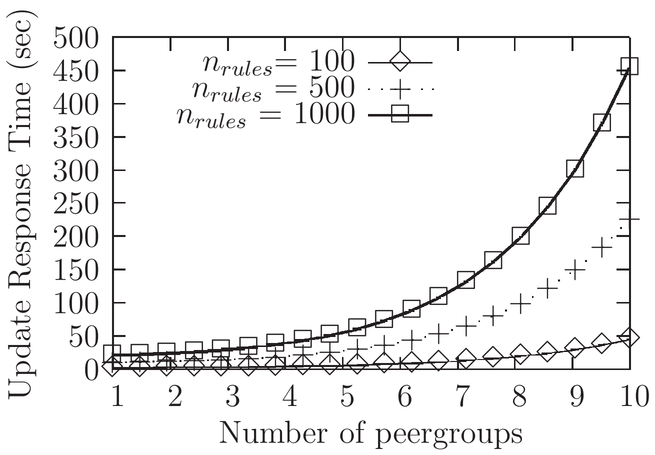


Figure 7.1: Model, Replication 10%, Full Net

values of $n$. This is to be expected as the presence of data in more peergroups increases the number of peers that a rule which has fired can be executed on, thus increasing the network traffic and repository load, and the number of further recursive rule firings.

Using the HyperCup topology in our system guarantees that each superpeer receives a message only once; if the number of superpeers is $n$, a total of $n - 1$ hops are required to reach all superpeers; and the most distant superpeers are reached after $\log_2 n$ hops.

Figure 7.2 shows,using the HyperCup topology, how the update response time varies as the number of peergroups $n$ increases, with the data replication $p_s$ being set to 0.1 and the number of rules per rule base, $n_{rules}$, set to 100, 500 and 1000. We see that the system now shows good scalability, and the update response time increases linearly with $n$.

Compared with the random topology, the system remains stable and the update response time within reasonable boundaries even for large networks and numbers of rules — given that in popular P2P systems, such as Kazaa and Gnutella,

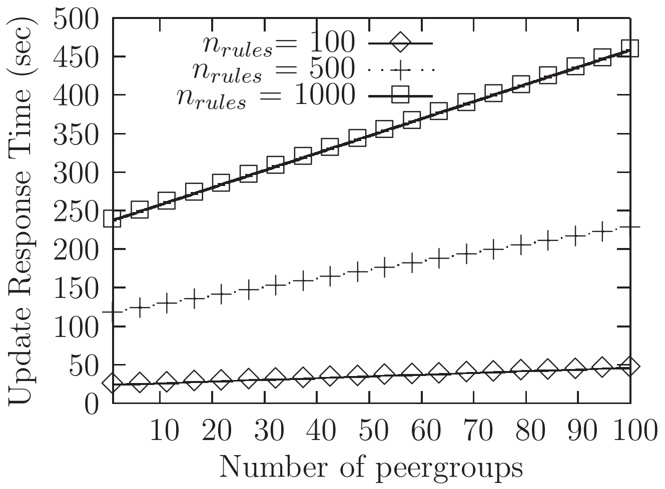


Figure 7.2: Model, Replication 10%, HyperCup

it sometimes takes minutes for a search request to complete. As with the random topology, increasing the data replication, $p_s$, increases the average update response time. However, in the case of HyperCup, the system still remains stable and the update response time still increases linearly even for high values of $p_s$. For example Figure 7.3 illustrates the case of $p_s = 0.5$ and Figure 7.4 illustrates the case of $p_s = 0.9$.

Figure 7.5 shows how the update response time varies with the HyperCup topology as the number of rules per rule base, $n_{rules}$ increases, with the data replication $p_s$ being set to 10% and the number of peergroups set to 10, 50 and 100. We see that the system again shows good scalability, with the update response time increasing linearly with $n_{rules}$.

For the analytical model, we have used values up to $n = 10,000$ and $n_{rules} = 10,000$, with varying $p_s$, and the same linear trends are observed.

Similar sets of experiments with different settings of the fixed parameters have also been conducted and this affects only the absolute performance values, and not the performance trends, with one exception: Varying $p_{reduct}$ (the reduction

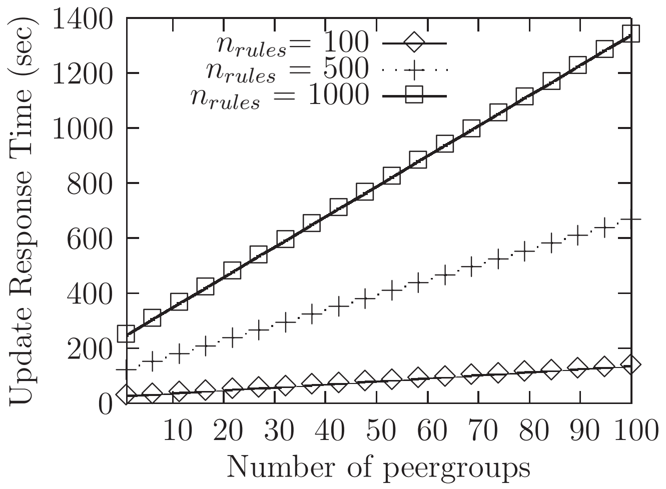


Figure 7.3: Model, Replication 50%, HyperCup

factor at each triggering level of the 'may-be-triggered' probability) between 0 and approximately 0.5 does not affect the performance trends of the system. However, for values above that, it gives similar performance trends only for values of $p_s$ (the level of data replication) of up to about 0.4. For higher values of $p_s$, the update response time becomes non-linear as the value of $n_{rules}$ or $n$ increases. This is because the high data replication results in more rules being triggered, more instances of rule actions being transmitted over the network and hence an increase in network transmission times. Moreover, the arrival rate of updates at the queues becomes higher than the rate that they can be served, causing significant increases of the queue waiting times. Varying $p_{mt}$ (the probability that a rule may be triggered, is also a measure of the selectivity of the index) between 0.1 and 0.9, we observe that with $p_{mt}$ values up to 0.3 the system shows good scalability characteristics and the update response time increases linearly. As the value of $p_{mt}$ increases above that (less selective index), the benefits of the index start to disappear and the system shows a non-linear behaviour that becomes progressively more severe.

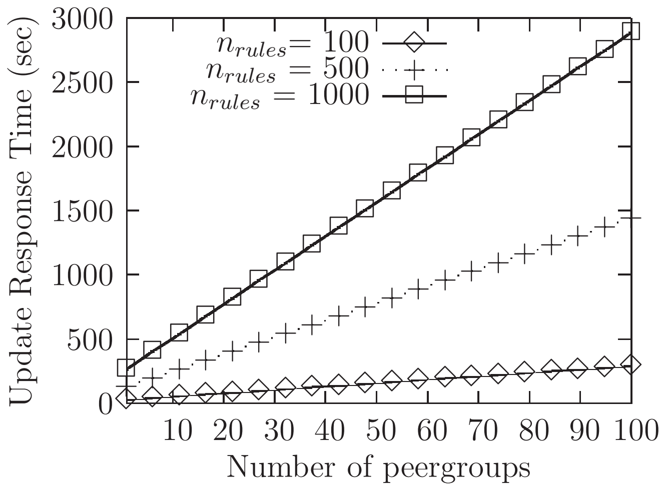


Figure 7.4: Model, Replication 90%, HyperCup

### 7.3.2 Comparison with a hard-coded approach

As already discussed in the context of XML in Section 4.5.2, an alternative to the ECA rule approach for providing reactive functionality in an application would be to hard-code the reactive functionality into the application. For example, referring back to Figure 2.4, a possible way to implement the requirement that users need to be notified of the registration of new LOs which satisfy a set of criteria that each user has specified, is to maintain a database, *db*, of users' 'subscriptions' to LOs.

The application code which handles the registration of new LOs would need to be modified so that it performs a look-up into this *db* and notifies each matching user of the new LO. As with the ECA rules approach, users would be able to change their set of subscription criteria: with the ECA approach, there would be a set of ECA rules per user and rules could be added to or deleted from this set, while with the hard-coded approach the subscriptions database *db* would be updated.

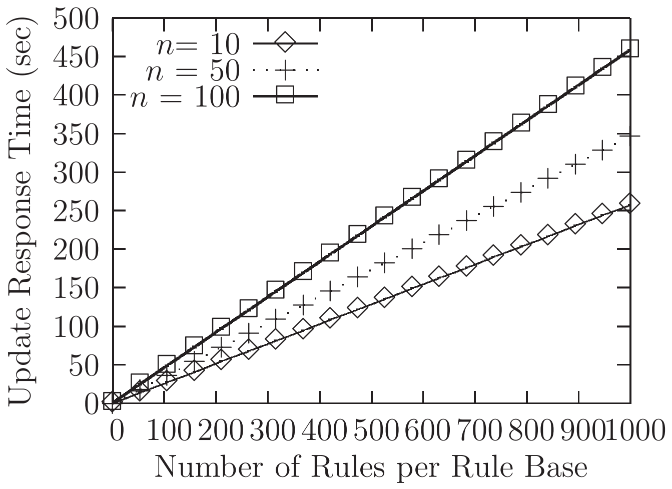


Figure 7.5: Varying Rules; Model, Replication 10% , HyperCup

Consider, for example, the RDFTL rule $r_1$ of Section 5.2.1 which notifies user 128, by updating their `newLO` metadata, of the insertion of a new LO whose subject is the same as one of user 128's areas of interest:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/user
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(sl_user:messages) IN
   INSERT ($msgs,sl_user:newLO,$delta);;
```

The corresponding hard-coded solution would implement this functionality in the application code, and the code snippet that would handle the registration of a new LO is described by the following pseudocode:

```
insertNewLO(newLO);
users[] := getSubscribedUsers(db);
foreach user in users[] do
    foreach interest in user.getInterests() do
        if (newLO.getSubject() == interest)
            updateUserProfile(user,newLO);
    end foreach;
end foreach;
```

Our analytical study above of the RDFTL rule processing system shows good performance and scalability characteristics when the HyperCup network topology is employed.

In the case of hard-coded reactive functionality, as in Section 4.5.2 for the XML case, we see that the cost of evaluating the condition and the action part of the rules is the same as in the ECA rule approach but there is again no event detection cost (which, in a P2P scenario, includes the time consuming messages between peers and their supervising superpeer).

We therefore modify our analytical model to remove the cost of event detection by setting $\overline{R}_{event}(i)$ shown in equation 7.10 to zero. The update response time for the hard-coded approach in a random topology with 10% data replication is shown in Figure 7.6. Comparing this with the corresponding graph for the ECA rules approach in Figure 7.1, we observe that, again, the system does not scale well. As the number of peergroups increases, the update response time rapidly rises towards very high values, but it shows lower update response time values for the same number of peergroups than the ECA approach.

Performing the same experiments for the hard-coded approach in a HyperCup topology — see Figures 7.7, 7.8, 7.9, 7.10 — we observe that, as for the ECA approach, the update response time increases linearly with the number of

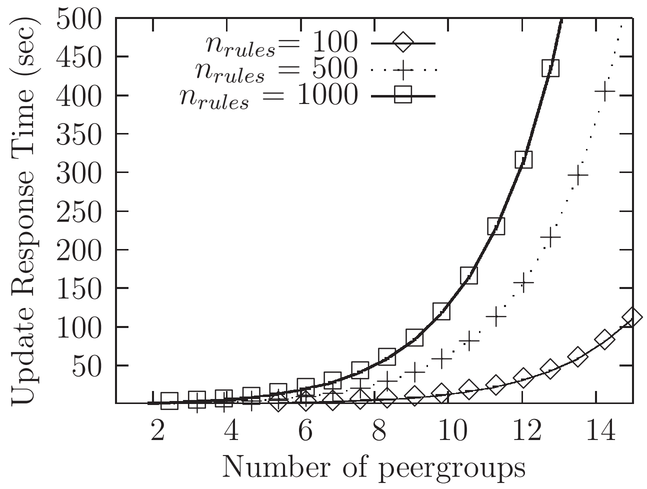


Figure 7.6: Model, Replication 10%, Full Net — Hard-Coded Case

peergroups, $n$. Lower update response time values are observed for the same $n$ compared with the ECA rules approach, and this can be explained by the lack of event detection overhead.

Although the performance behaviour of the two approaches shows similar performance trends, the lower update response time of the hard-coded approach would be an attractive feature for performance-critical applications that do not have a large variety of rules to encode and maintain. In cases, however, that an application is not performance-critical and makes extensive use of a broad variety of reactive functionalities, the ECA rules approach would be more attractive: rules could be dynamically added or deleted while the application is still running, and the system would be likely to be more robust to programmer errors.

### 7.3.3 The Simulator

As well as developing the performance model discussed above, we have also developed a simulator of the RDFTL system in order to validate the predictions of the

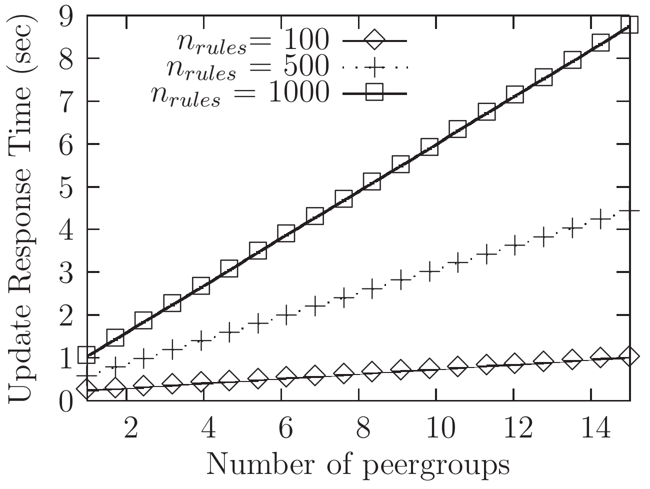


Figure 7.7: Model, Replication 10%, HyperCup — Hard-Coded Case

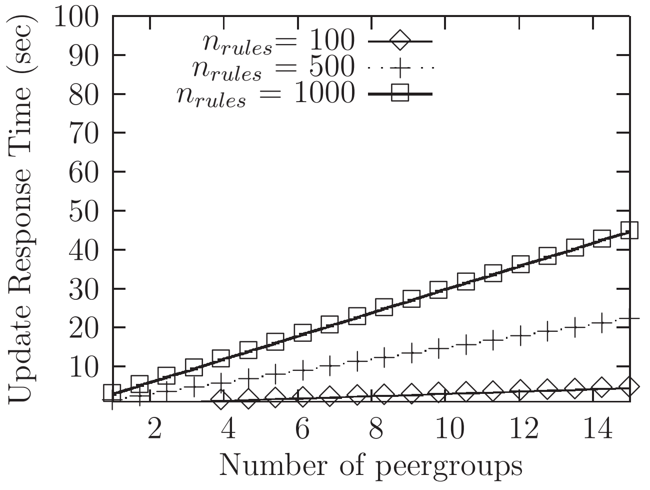


Figure 7.8: Model, Replication 50%, HyperCup — Hard-Coded Case

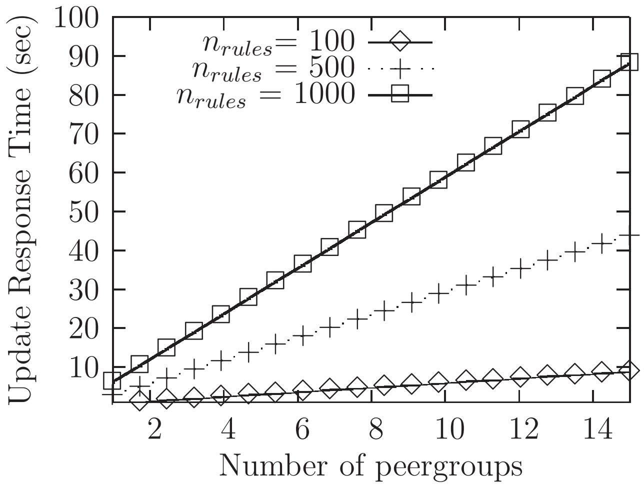


Figure 7.9: Model, Replication 90%, HyperCup — Hard-Coded Case

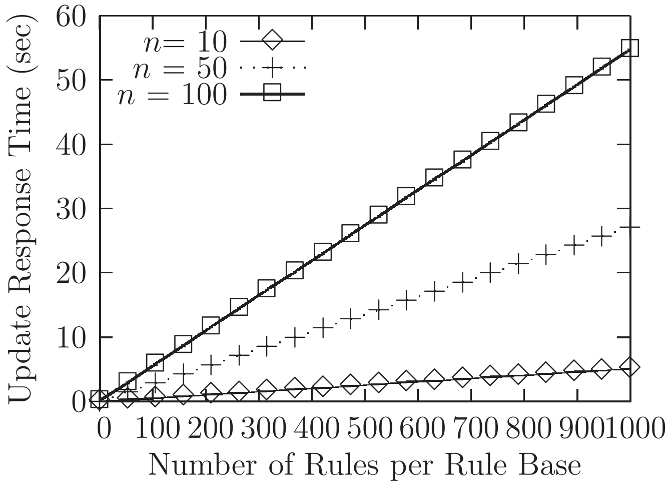


Figure 7.10: Varying Rules; Model, Replication 10% , HyperCup — Hard-Coded Case

analytical model. This simulator was built using the Java implementation of the SSF (Scalable Simulation Framework) API [SSF], called Raceway SSF [R-SSF]. Raceway SSF provides a unified interface for discrete-event simulation, consisting of a set of classes for modeling the system entities, simulation events, communication channels and processes according to the discrete-event paradigm. The peers, superpeers, rule bases and rules are the main entities of our simulator. Peers and superpeers exchange messages representing events that can potentially affect the system state. For example, a message sent from a peer to a superpeer notifying the superpeer of the occurrence of an update at the peer may cause rule triggering that may result in more updates being executed, and so forth.

Each component of the actual system implementation is represented by a corresponding class in the simulator. Some of these components, particularly those implementing the rule processing logic, have been used directly in the simulator, while others have been replaced by simulated functionality. In particular, the Event Handler, Condition Evaluator and Action Scheduler components have remained intact, although some wrapping code was necessary to interface them with the simulated components. The rest of the components and services shown in Figure 6.5, have been replaced by simulated functionality. Other system entities, such as the peers and superpeers, network connections and topology, and the Rule Base have also been replaced and are simulated by program classes. When the simulator initialises, it creates a pre-specified number of SuperPeer objects and "connects" to each of them a number of Peer objects that are recorded in the SuperPeer object. Which other SuperPeer objects each SuperPeer is connected with depends on the specified network topology and this information is also recorded in the SuperPeer object.

For the simulated components, their behaviour is simulated by a set of functions each of which represents an action taken or a service provided by the component. Fixed parameters such as the network delay, number of rules in a rule base, or number of actions per rule, are represented by variables in the class corresponding to the relevant component. A set of statistical distributions is used to simulate the behaviour of the other system parameters and to obtain their values according to the distributions. For example, query service time and transaction interarrival time are exponentially distributed, while the rule triggering probability is normally distributed. Each time that a value of one of these parameters is needed, a random number generator undertakes to generate a new random value according to the given distribution.

The services in the Core Services group of Figure 6.5 are each simulated by a function in the Peer or SuperPeer class, as follows: The Peer Indexing Service is simulated by a random number generator function that returns 0 or 1 – representing the probability that the index contains the requested data – distributed around a mean value. The Database Connection Service is simulated by two functions, one for updates and one for queries. Calling each of these functions adds a random time delay — corresponding to the database processing time — that is derived by calling a random number generator producing numbers about a mean value. The Messaging Service is simulated by a function that (i) adds a time delay, corresponding to network transmission time, produced by a random number generator around a chosen mean; and (ii) calls the SuperPeer object's Event Handler (see below). The Event Detection service is simulated by a set of functions and classes that simulate the event detection functionality in the Peer and SuperPeer classes. Each time that a call to the data update function in a Peer or SuperPeer class takes place, a notificaction is produced calling the function corresponding to the messaging service.

The services in the RuleBase Management Services group of Figure 6.5 are also simulated by functions in the SuperPeer class, while the Rule Base is simulated by a class of its own. The Rule Base Indexer Service is simulated by a function that returns a number of rules, which ranges from 0 to the number of rules recorded in the Rule Base class, exponentially distributed. The Rule Base class has as its main parameter the number of rules in this superpeer's rule base, and contains methods for setting and retrieving the number of rules. The Rule Base Registration Service is omitted in the simulator as this component is not involved in rule execution.

The SuperPeer Indexing Service is simulated in the same way as the Peer Indexing Service. Finally, the Routing Service at each superpeer is simulated by a look-up into the SuperPeer's list of connections to other SuperPeer and Peer objects it is connected with.

An internal clock, initilised and managed by the simulation application, is used for recording the number of time units elapsed for executing each function of the simulation application. The simulation starts when external update transactions arrive, at times that are exponentially distributed, at Peer and Superpeer objects, and ends after the end of the execution of all the actions caused by the arrival of the last external update transaction. A specified number of such external transactions is executed per simulation run, randomly submitted to the Peer and Superpeer obects.

### 7.3.4 Simulation Results

The same set of experiments performed on the analytical model were performed with the simulator. In these experiments, the same values for $bps$, $k$ and $p_{reduct}$ were used as for the analytical model experiments, as shown in Table 1. The remaining system parameters of Table 1 were replaced by stochastic values in order to provide a more realistic description of the system. In particular, the number of peers per peergroup, $m$, is randomly distributed with a mean value of 20; the probabilities $p_{allow}$, $p_t$ and $p_f$ are normally distributed, with mean values as in Table 1; and the remaining parameters are exponentially distributed, with mean values as in Table 1. Similarly, in the experiments, the number of rules per rule base, $n_{rules}$, and the degree of replication, $p_s$ are not fixed but are normally distributed about a chosen mean.

The distributions of the various parameters may vary for different applications and data sets, resulting in potentially different absolute performance values, though we conjecture that it will not affect the performance trends of the system. This is an area of further investigation.

Each of the data points in the graphs below was obtained by running the simulator for 2000 top-level updates and taking the average update response time. The experiments were again conducted with the random and with the HyperCup network topologies.

Figure 7.11 shows how the update response time varies with the random topology as the number of peergroups, $n$, increases, with the data replication $p_s$ distributed about a mean value of 0.1 and the number of rules per rule base, $n_{rules}$, distributed about mean values of 100, 500 and 1000. We see that the trend of these graphs is similar to Figure 7.1 from the analytical study, and it indicates that the system does not scale well with increasing $n$. Similar sets of experiments conducted for higher average values of $p_s$ (up to 0.9) result in graphs with similar upwards trends, except that the absolute values of the update response time are now larger and the system becomes unstable at lower values of $n$.

Results from the same set of experiments using the simulator and assuming a HyperCup topology are shown in Figures 7.12 , 7.13 and 7.14. We see that

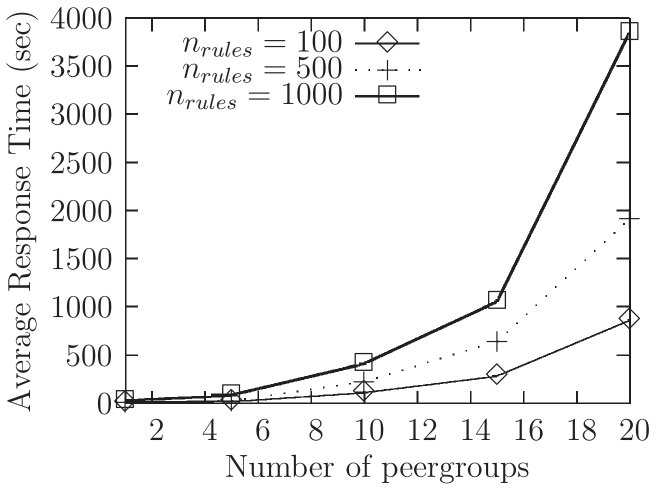


Figure 7.11: Simulation, Replication 10%, Full Net

the system now shows good scalability, with the update response time increasing approximately linearly with $n$. With data replication at 10%, the update response time is within reasonable boundaries even for relatively large networks and numbers of rules. The system appears to scale well even for higher levels of data replication as illustrated in Figure 7.14 for data replication at 90%.

Finally, Figure 7.15 shows how the update response time varies with the HyperCup topology as the average number of rules per rule base, $n_{rules}$ increases, with the data replication $p_s$ being set to an average of 10% and the number of peergroups set to 10, 50 and 100. We see that the system again shows good scalability, with the update response time increasing linearly with $n_{rules}$.

The difference in the absolute values of the update response time between the analytical model and the simulations can be explained by the fact that in the analytical model we have used fixed values for the system parameters of Table 7.3 while in the simulations the values of these parameters vary following some distribution. The results from the simulations are thus likely to be closer to the real system applications, but their trends shown good agreement with those

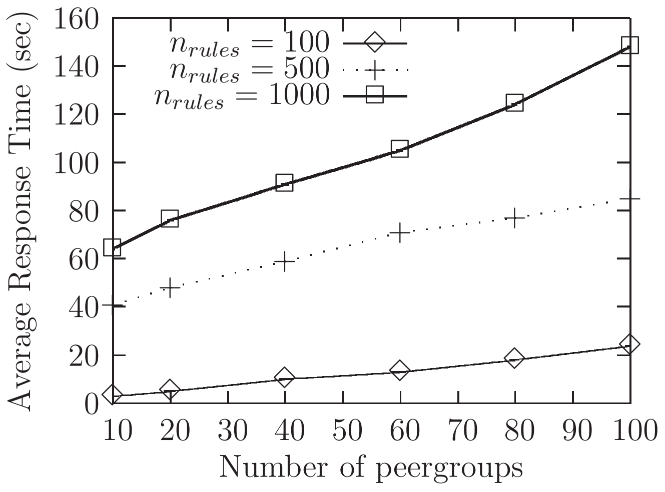


Figure 7.12: Simulation, Replication 10%, HyperCup

of the analytical model.

## 7.4 Summary

This chapter has studied the performance and scalability aspects of processing RDFTL rules on RDF metadata in P2P environments, as implemented in the system presented in Chapter 6. We have developed and described an analytical model for this system and we have presented the experimental results of an analytical study to examine how system performance is affected by factors such as the network topology, the number of peers, and the degree of RDF data replication between peers. We have also developed and described a simulation of the system, and have conducted similar performance experiments with the simulator. The two sets of experimental results show good agreement, which is an indication of the validity of the analytical performance model. To our knowledge, this is the first time that a P2P ECA rule processing system has been studied from a performance perspective, and moreover we have employed both analytical and

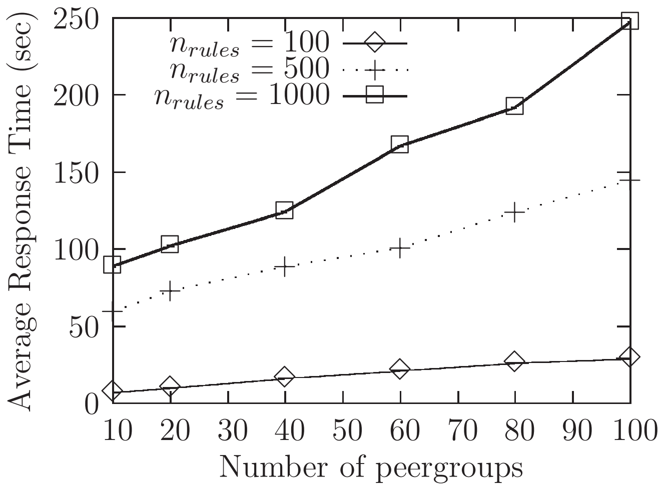


Figure 7.13: Simulation, Replication 50%, HyperCup

simulation methods.

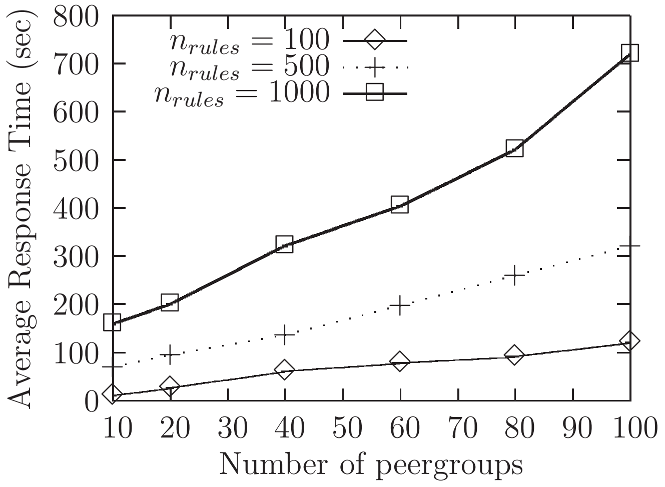


Figure 7.14: Simulation, Replication 90%, HyperCup

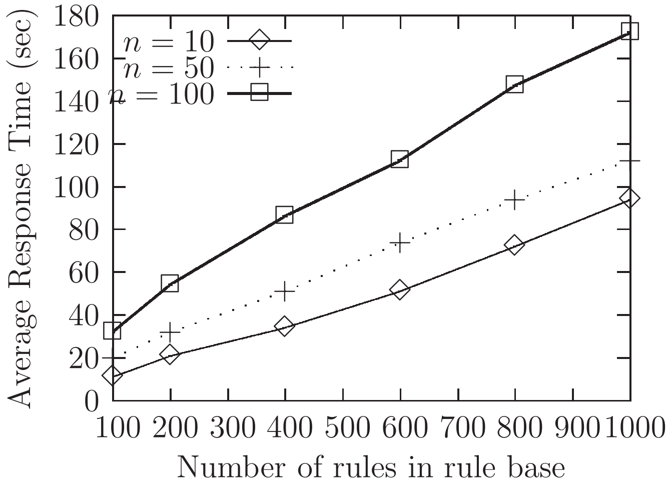


Figure 7.15: Varying Rules; Simulation, Replication 10%, HyperCup

# Chapter 8

# Conclusions and Future Work

A combination of formal and empirical approaches have been used as the methodology for the research reported in this thesis. This thesis has discussed the use of ECA rules for providing reactive functionality over XML and RDF data in both centralised and distributed environments. We have reviewed an existing ECA rule language for XML data called XTL and have proposed a new ECA rule language for RDF data called RDFTL. We have studied the expressiveness of XTL. We have described the architecture and the implementation of two prototype systems for processing XTL and RDFTL rules in centralised and distributed environments, respectively. We have developed analytical performance models for both systems and have compared these with the performance of the actual system in the case of XTL and of a system simulation, in the case of RDFTL. We have also discussed the extent to which RDFTL meets the SeLeNe requirements regarding reactive functionality.

During our research we have considered the following four research problems: what should be the constructs and features of ECA languages in order to support the definition and processing of ECA rules over XML and RDF data; what are the architectural requirements of systems supporting ECA rule processing over

XML and RDF data in centralised and P2P environments; what are the factors that affect the performance of such ECA rule processing systems; and what are the performance trends and the scalability characteristics of such systems.

In Chapter 2, we gave an overview of ECA rule languages for semi-structured data including a review of the query and update languages that may be used in the different parts of ECA rules.

In Chapter 3, we reviewed the XTL ECA rule language for XML data. We conducted a study of the expressiveness of XTL and showed that it is able to express programs at least as complex as *while* programs over relational data. We also showed that the extension of the language to support *let* expressions in the action part of rules and functions for creating new unique values makes the language query complete over relational data.

In Chapter 4, we described a prototype system that supports the definition and processing of XTL rules in a centralised environment. We presented its architecture, its major components and the interaction between them. We also conducted a performance study of the system using both analytical methods and experiments on the system itself. The study showed the system performance trends and we discussed possible reasons for performance penalties. We showed that the provision of an indexing scheme for XTL rules could lead to a significant improvement in system performance.

In Chapter 5, we presented the RDFTL ECA rule language for RDF data, including its syntax, the denotational semantics of its query and update sublanguages, and the syntax of RDFTL rules.

In Chapter 6, we described a prototype system that supports the definition and processing of RDFTL rules in P2P environments. We presented its architecture, its major components and the interaction between them. We described how rules are registered and executed in such a P2P environment, and we discussed possible approaches to concurrency control and recovery in such environments. We also showed the extent to which RDFTL meets the SeLeNe requirements regarding reactive functionality.

Although our RDFTL rule processing system has been developed for a P2P environment, it can be easily modified to operate in any distributed architecture, or in a centralised environment similar to our XTL rule processing system. Conversely, the P2P infrastructure that we developed to support RDFTL rules can be easily extended so as to be used by other ECA rule processing systems, for example our XTL rule processing engine, and this is an area of ongoing work.

In Chapter 7, we conducted a study of the performance and scalability of our RDFTL rule processing system. We developed a performance model based on analytical methods and we investigated the system's performance using experimental results from both the analytical model and system simulations. We showed that the performance of the system depends on the number of messages that are exchanged during rule processing. This number is highly dependent on the network topology employed, and we showed that if a HyperCuP topology is used for interconnecting the superpeers then our system shows good scalability. This points to the practical usefulness of our RDFTL rule processing system for real P2P applications.

Some of the results presented in this thesis have been published in [BPPW04, PPW03b, PPW04, PPW06]. The XTL rule processing system of Chapter 4 was described in [BPPW04]. An early version of the RDFTL language was described in [PPW03b] and its final version as presented in Chapter 5 was described in [PPW04]. The RDFTL rule processing system of Chapter 6 was described in [PPW06].

There are several directions of future research that build on the results of this thesis:

1. Extend the XTL language and system implementation

   Extending the XTL action sublanguage to support a richer set of update expressions will improve the language's update capabilities over XML data, for example, to support ordering in XPath expressions and to adopt the emerging XQuery Update Facility proposal [CFR06]. The XTL system implementation can be extended to enable support for XML repositories and to implement our XTL rule indexing scheme. As discussed in Chapter 4, we expect that these extensions will significantly improve the system's performance.

2. Extend and improve the RDFTL system implementation

   Possible next steps regarding the RDFTL system would be to add support for the Jena2 RDF management framework and its accompanying query language, SPARQL [W3C06c]. The current communication layer can be replaced with a more standard communication protocol, such as SOAP, in order to make the system more compatible with existing and emerging distributed service-based architectures and systems. This will enable future extensions of system with third-party components such as distributed query and transaction management components that will in turn enable the support of distributed event detection and update execution.

3. Extend the performance study of the XTL and RDFTL rule processing systems

   The performance study for both the XTL and RDFTL systems can be extended in order to explore additional performance parameters, such as CPU load, memory consumption, number of I/O operations and network usage. This will give a broader view of our systems' performance and scalability characteristics.

4. Conduct large-scale experiments with the RDFTL system itself, possibly using the PlanetLab [PlanetLab] infrastructure. As well as giving insight into the actual system behaviour in a real P2P environment, this will allow measurements on actual system workloads and rule sets, which can then be fed into the analytical performance model and the simulator to allow more accurate predictions from these.

5. Further study of XTL's and RDFTL's expressiveness.

   Further study into RDFTL's query expressiveness is required, and further study into the update expressiveness of both languages. Regarding the latter, making the language relationally update complete requires extending them with the ability to create new values. However, this ability can lead to *unsafe*[1] conditions, thus requiring more investigation regarding the characteristics of the language that will enable update completeness but will also ensure safety.

6. Analysis and Optimisation of XTL and RDFTL rules

   In [BPW02c] techniques are proposed for analysing the behaviour of XTL rules and optimising such rules is also proposed. Similar techniques need to be developed for RDFTL. Extending the XTL and RDFTL rule processing systems with rule optimisation techniques is an open research issue, which is related to further study of the XML and RDF data models and the optimisation of query and update languages for these models.

7. Application of XTL and RDFTL in the e-Learning domain

   Our XTL and RDFTL rule processing systems can be deployed as part of e-learning applications, allowing further experimentation with real data

[1]Unsafe queries are queries that have an infinite number of answers. Unsafety is caused by variables occurring in the output of a query which are not bounded by the input to the query.

and rule sets. This will enable the collection of information regarding the systems' behaviour in real conditions that may improve the accuracy of our analytical models and lead to research in further optimisation techniques in order to improve system behaviour.

# Bibliography

[4Suite] 4Suite Consortium. 4Suite XML Tools. `http://4suite.org/`.

[ABEYH00] A. Adi, B. Botzer, O. Etzion, and T. Yatzkar-Haham. Push Technology Personalisation through Event Correlation. In *Proceedings of the 26th Conference on Very Large Databases (VLDB)*, pages 643–645, 2000.

[ABS00] S. Abiteboul, Peter Buneman, and Dan Siciu. *Data on the Web*. Morgan Kaufmann, 2000.

[AV91] Serge Abiteboul and Victor Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer Systems Science*, 43(1):62–124, 1991.

[AVFY98] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles Of Database Systems (PODS)*, pages 179–187, 1998.

[AVH96] Serge Abiteboul, Victor Vianu, and Richard Hull. *Foundations of Databases*. Addison Wesley, 1996.

[BB97] Elena Baralis and Andrea Bianco. Performance Evaluation of Rule Semantics in Active Databases. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 365–374, 1997.

[BBCC02] Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 403–413, 2002.

[BBFV05] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding Updates to XQuery: Semantics, Optimisation and Static Analysis. In *Proceedings of the 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2005.

[BCP00] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active rules for xml: A new paradigm for e-services. In *Proceeding of the Workshop on Technologies for E-Services (TES 2000)*, pages 77–94, 2000.

[BCP01a] A. Bonifati, S. Ceri, and S. Paraboschi. Active Rules for XML: A new paradigm for e-Services. *VLDB Journal*, 10(10):39–47, 2001.

[BCP01b] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. In *Proceedings of the 10th World Wide Web Conference*, pages 633–641, 2001.

[BDZH95] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of data*, page 476, 1995.

[Bec04] Dave Beckett. RDF/XML Syntax Specification. `http://www.w3.org/TR/rdf-syntax-grammar/`, February 2004.

[BEPR06] Francois Bry, Michael Eckert, Paula-Lavinia Patranjan, and Inna Romanenko. Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2006)*, 2006.

[BFKM85] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[BLCG92] Tim Berners-Lee, Robert Cailliau, and Jean-François Groff. The World-Wide Web. *Computer Networks and ISDN Systems*, 25(4-5):454–459, 1992.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commununications ACM*, 13(7):422–426, 1970.

[BPC01] James Bailey, Alexandra Poulovassilis, and Simon Courtenage. Optimising Active Database Rules by Partial Evaluation and Abstract Interpretation. In *Proceedings of the 8th International Workshop of Database Programming Languages (DBPL)*, pages 300–317, 2001.

[BPF$^+$94] M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In *Proceedings of the 20th International Conference of Very Large Databases (VLDB)*, pages 463–474, 1994.

[BPF$^+$95] M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. ROCK & ROLL: A Deductive Object-Oriented Database System. *Information Systems*, 20:185–211, 1995.

[BPPW04] James Bailey, George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action Rule Language for XML. In *Web Dynamics*, pages 223–248. Springer-Verlag, 2004.

[BPW02a] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action language for XML. In *Proceedings of the 11th World Wide Web Conference*, pages 486–495, 2002.

[BPW02b] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. Analysis and Optimisation of Event-Condition-Action Rules on XML. *Computer Networks*, 39:239–259, 2002.

[BPW02c] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. Analysis and optimisation of Event-Condition-Action rules on XML. *Computer Networks*, 39(3):239–259, 2002.

[Bun97] Peter Buneman Semistructured Data. In *Proceedings of the 16th ACM Symposium of Principles of Database Systems (PODS)*, pages 117–121, 1997.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[CCR04] Miguel Castro, Manuel Costa, and Antony Rowston. Peer-to-Peer overlays: structured, unstructured, or both? Technical Report MSR-TR-2004-73, Microsoft Research, 2004.

[CFP99] S. Ceri, P. Fraternali, and S. Paraboschi. Data-driven one-to-one web site generation for data intensive applications. In *Proceedings of the 25th International Conference on Very Large Databases*, pages 615–626, 1999.

[CFR06] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery Update Facility. `http://www.w3.org/TR/xqupdate/`.

[CFR05] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery Update Facility Requirements. `http://www.w3.org/TR/xquery-update-requirements/`, June 2005.

[CGM02] Arturo Crespo and Hector Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *Proceeding of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 23–34, 2002.

[CGMR95] S. Castangia, Giovanna Guerrini, Danilo Montesi, and G. Rodriguez. Design and Implementation for the Active Rule Language of Chimera. In *Proceedings of the 6th International Conference on Database and Expert Systems Applications (DEXA)*, pages 45–54, 1995.

[CKK$^+$03] V. Christophides, G. Karvounarakis, I. Koffina, G. Kokkinidis, A. Magkanaraki, D. Plexousakis, G. Serfiotis, and V. Tannen. The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware. In *Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB)*, pages 381–393, 2003.

[CM94] Sharma Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[CPHK02] Eunsuk Cho, Insuk Park, Soon J. Hyun, and Myungchul Kim. ARML: an active rule mark-up language for heterogeneous active information systems. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002.

[CS02] Edith Cohen and Scott Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. *SIGCOMM Computer Communication Review*, 32(4):177–190, 2002.

[DB2XML] IBM. DB2 XML Extender. `http://www-306.ibm.com/software/data/db2/extenders/xmlext/`, 2002.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, 1988.

[DBLP06] Universitat Trier. DBLP Bibliography. `www.informatik.uni-trier.de/ ley/db/`.

[DBM88] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules Are Objects Too: A Knowledge Model for an Active Object Oriented Database System. In *Proceedings of the 2nd International Workshop on OODBMS*, pages 129–143, 1988.

[dbXML] dbXML: Native XML Database. `http://www.dbxmlgroup.com/`.

[DC03] Dublic Core Consortium. Dublic Core RDF Vocabulary. `http://purl.org/dc/elements/1.1/`, 2003.

[DJ97] Oscar Díaz and Arturo Jaime. EXACT: An Extensible Approach to Active Object-Oriented Databases. In *Proceedings of the 23rd International Conference of Very Large Databases (VLDB)*, pages 282–295, 1997.

[DPW99] Andrew Dinn, Norman W. Paton, and M. Howard Williams. RAP: The ROCK & ROLL Active Programming System. In *Active Rules in Database Systems*, pages 323–336. Springer-Verlag, 1999.

[eXist] eXist: Open Source Native XML Database. `http://exist.sourceforge.net/`.

[Freenet] Freenet. See. `http://www.freenet.sourceforge.com`.

[FW04] David C. Fallside and Priscilla Walmsley W3C. XML Schema Part 0: Primer Second Edition. `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[GD92] Stella Gatziu and Klaus R. Dittrich. SAMOS: an Active Object-Oriented Database System. *IEEE Data Engineering Bulletin*, 15(1-4):23–26, 1992.

[GD94] Stella Gatziu and K.R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS)*, pages 2–9, 1994.

[GGD95] A. Geppert, S. Gatziu, and K. R. Dittrich. A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, pages 309–326, 1995.

[GGM$^+$04] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Siciu. Processing XML Streams With Deterministic Automata and Stream Indexes. *ACM Transactions on Database System*, 29(4):752788, 2004.

[GJS92] N. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th International Conference of Very Large Databases (VLDB)*, pages 327–338, 1992.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[Gnutella] Gnutella. See. `http://www.gnutella.wego.com`.

[GWJD03] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating data sources in large distributed systems. In *Proceeding of the 29th International Conference of Very Large Databases (VLDB)*, pages 874–885, 2003.

[Han96] E. N. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):157–172, 1996.

[HSS03] K. Haller, H. Schuldt, and H.J. Schek. Transactional peer-to-peer information processing: The AMOR approach. In *Proceedings of the 4th International Conference on Mobile Data Management*, pages 356–362. Springer, 2003.

[IEEE-LOM] IEEE Learning Technology Standards Committee. IEEE Learning Object Metadata (LOM). `http://ltsc.ieee.org/wg12/`.

[IKK98] H. Ishikawa, K. Kubota, and Y. Kanemasa. XQL: A Query Language for XML Data. In *Proceedings of the W3C Workshop on Query Language*, 1998.

[IO01] H. Ishikawa and M. Ohta. An active web-based distributed database system for e-commerce. In *Proceedings of the 1st International Workshop on Web Dynamics*, 2001.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[JavaCC05] Java.net. Java Compiler Compiler (JavaCC) - The Java Parser Generator. `https://javacc.dev.java.net/`.

[Jelly] Apache Foundation. Jelly : Executable XML. `http://jakarta.apache.org/commons/jelly/`.

[Jena2] Hewlett-Packard Labs. Jena Semantic Web Framework. `http://jena.sourceforge.net/`.

[KAC$^+$02] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *Proceedings of the 11th International World Wide Web Conference*, pages 592–603, 2002.

[Kazaa] Kazaa. See. `http://www.kazaa.com`.

[KC04] Giorgos Kokkinidis and Vassilis Christophides. Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In *Proceeding of the 9th International Conference on Extending Database Technology (EDBT)*, pages 486–495, 2004.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pages 95–106, 1990.

[KMC99] Krishna G. Kulkarni, Nelson Mendonça Mattos, and Roberta Cochrane. Active Database Features in SQL3. In *Active Rules in Database Systems*, pages 197–219. 1999.

[KP03] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, pages 29–47, 2003.

[KPPL03] K. Keenoy, G. Papamarkos, A. Poulovassilis, M. Levene, D. Peterson, P.T. Wood, and G. Loizou. WP2 Deliverable 2.2: Self E-Learning Networks – Functionality, User Requirements and Exploitation Scanarios Technical report, School of Computer Science and Information Systems, Birkbeck, University of London, 2003.

[LM00] Andreas Laux and Lars Martin. XML:DB — XUpdate Working Draft.

http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, September 2000.

[MP03] P. McBrien and A. Poulovassilis. Defining Peer-to-Peer integration using Both-As-View rules. In *Proceedings of the 1st International Conference on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, pages 91–107, 2003.

[MSCK05] M. Magiridou, S. Sahtouris, Vassilis Christophides, and Manolis Koubarakis. RUL: A declarative update language for RDF. In *Proceedings of the 4th International Semantic Web Conference*, pages 506–521, 2005.

[MSSQL05] Microsoft Corporation. SQL Server 2005. http://www.microsoft.com/sql/default.mspx, 2005.

[NJ00] Matthias Nicola and Matthias Jarke. Performance Modeling of Distributed and Replicated Databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, 2000.

[NSST02] Wolfgang Nejdl, Wolf Siberski, Bernd Simon, and Julien Tane. Towards a Modification Exchange Language for Distributed RDF Repositories. In *Proceedings of the 1st International Semantic Web Conference*, pages 236–249, 2002.

[NWQ$^+$02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjorn Naeve, Mikael Nilsson, Matthias Palmer, and Tore Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *Proceedings of the 11th International World Wide Web Conference*, pages 604–615, 2002.

[NWS+03] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario T. Schlosser, Ingo Brunkhorst, and Alexander Löser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proceedings of the 12th International World Wide Web Conference*, pages 536–543, 2003.

[OQL01] Object Data Management Group. Object Query Language (OQL). `http://www.odmg.org/`.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proceedings of the Workshop on XML Data Management (XMLDM)*, pages 109–127, 2002.

[OraXML] Oracle. Oracle XML DB 10g. `http://www.oracle.com/technology/tech/xml/xmldb/index.html`.

[Pat89] Norman W. Paton. ADAM: An Object-Oriented Database System Implemented in Prolog. In *Proceedings of the 7th British National Conference on Databases*, pages 147–161, 1989.

[Pat99] Norman W. Paton. *Active Rules In Database Systems.* Springer, 1999.

[PlanetLab] PlanetLab. `http://www/.planet-lab.org`.

[PPW03a] G. Papamarkos, A. Poulovassilis, and P. T. Wood. WP4 Deliverable 4.4 — ECA Rules Languages for Active Self e-Learning Networks. Technical report, School of Computer Science and Information Systems, Birkbeck, University of London, 2003.

[PPW03b] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In

*Proceedings of the 1st International Workshop on Semantic Web and Databases*, pages 309–327, September 2003.

[PPW04] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proceedings 3rd Web Dynamics Workshop at WWW'2004*, pages 223–248, 2004.

[PPW06] George Papamarkos, Alex Poulovassilis, and Peter T. Wood. Event-Condition-Action Rules on RDF Metadata in P2P Environments. *Computer Networks*, 50(10):1513–1532, 2006.

[PS91] Alexandra Poulovassilis and Carol Small. A functional programming approach to deductive databases. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pages 491–500, 1991.

[PV97] Philippe Picouet and Victor Vianu. Expressiveness and Complexity of Active Databases. In *Proceedings of the 6th International Conference on Database Theory*, pages 155–172, 1997.

[R-SSF] Raceway SSF. `https://gradus.renesys.com/exe/Raceway`.

[RCF00] Jonathan Robie, Don Chamberlin, and Daniela Florescu. QUILT: an XML Query Language. `http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html`, March 2000.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. pages 329–350, 2001.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 161–172, 2001.

[RDFPath] RDFPath Query Language. `http://logicerror.com/RDFPath`.

[RDFSuite] ICS FORTH. Institute of Computer Science of the Foundation for Research and Technology , RDFSuite. `http://139.91.183.30:9090/RDF/`.

[RPS95] Swarup Reddi, Alexandra Poulovassilis, and Carol Small. Extending a Functional DBPL with ECA-rules. In *Proceedings of the 2nd International Workshop on Rules in Database Systems (RIDS)*, pages 101–118, 1995.

[RS03] Philippe Rigaux, and Nicolas Spyratos. Generation and Syndication of Learning Object Metadata. Laboratoire de Recherche en Informatique, Universite Paris-Sud Orsay, France, 2004.

[RuleML] The Rule Markup Initiative. `http://www.ruleml.org/`.

[SeLeNe] SeLeNe : Self e-Learning Networks. `http://www.dcs.bbk.ac.uk/selene/`.

[Sesame] openRDF Consortium. Sesame: RDF Schema Querying and Storage. `http://www.openrdf.org/`.

[SC04] The SeLeNe Consortium. An Architectural Framework and Deployment Choices of SeLeNe. Technical report, School of Computer Science and Information Systems, Birkbeck, University of London, 2004.

[SHP88] Michael Stonebraker, Eric N. Hanson, and Spyros Potamianos. The postgres rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, 1988.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 149–160, 2001.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[SSDN02] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP – hypercubes, ontologies and efficient search on P2P networks. In *Proceedings of the 1st International Workshop on Agents and P2P Computing*, pages 112–124, 2002.

[SSF] SSF Scalable Simulation Framework. `http://www.ssfnet.org/homePage.html`.

[Starburst] IBM Research Labs. Starburst. `http://www.almaden.ibm.com/cs/starwinds/starburst.html`, 1992.

[ST04] Wolf Siberski and Uwe Thaden. A Simulation Framework for Schema-Based Query Routing in P2P Networks. In *Proceedings of the 1st Workshop on Peer-to-Peer Computing and Databases* , pages 436–445, 2004.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 413–424, May 2001.

[TIM$^+$03] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The Piazza Peer Data Management Project. *SIGMOD Record*, 32(3):47–52, 2003.

[TJO01] Berners-Lee T., Handler J., and Lassila O. The Semantic Web. *Scientific American*, May, 2001.

[OV99] M. Tamer Ozsu, Patrick Valduriez *Principles of Distributed Database Systems.* Prentice Hall, 1999.

[UDDI] UDDI Organization. UDDI. `http://www.uddi.org/`.

[Versa] Uche Ogbuj. Versa RDF Query Language. `http://uche.ogbuji.net/tech/rdf/versa/`.

[W3CDOM] W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`.

[W3C99a] W3C. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/2005/CR-xpath20-20051103/`, November 1999.

[W3C99b] W3C. XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`, November 1999.

[W3C01] W3C. XML Pointer Language (XPointer) Version 1.0. `http://www.w3.org/TR/WD-xptr`, January 2001.

[W3C04a] W3C. RDF Semantics, W3C Recommendation 10 February 2004, 2004.

[W3C04b] W3C. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004, 2004.

[W3C04c] W3C. RDF/XML Syntax Specification, W3C Recommendation 10 February 2004, 2004.

[W3C04d] W3C. RDQL - A Query Language for RDF. http://www.w3.org/Submission/RDQL/, January 2004.

[W3C04e] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/, February 2004.

[W3C05a] W3C. XML Path Language (XPath) Version 2.0. http://www.w3.org/TR/2005/CR-xpath20-20051103/, November 2005.

[W3C05b] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, November 2005.

[W3C06a] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/REC-xml/, August 2006.

[W3C06b] W3C. SPARQL Protocol for RDF. http://www.w3.org/TR/rdf-sparql-protocol/, January 2006.

[W3C06c] W3C. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, February 2006.

[W3C06d] W3C. XML Schema 1.1. http://www.w3.org/TR/xmlschema11-1/, March 2006.

[Wag02] Gerd Wagner. How to Design a General Rule Markup Language? In *Invited talk at the Workshop XML Technologien für das Semantic Web (XSW 2002), Berlin*, pages 19–37, June 2002.

[WBT92] D.L. Wells, J.A. Blakeley, and C.W Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–82, 1992.

[WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems — Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[Wid92] Jennifer Widom. The Starburst Rule System: Language Design, Implementation, and Applications. In *IEEE Data Engineering Bulletin*, volume 15, pages 15–18. 1992.

[Wid96] Jennifer Widom. The Starburst Rule System. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 87–109. Morgan Kaufmann, 1996.

[Xindice] Apache Foundation. Xindice XML Repository. http://xml.apache.org/xindice/.

[XML:DB] XML:DB Initiative for XML Databases. http://xmldb-org.sourceforge.net/.

[XMLRPC] XML-RPC.com. XML-RPC : Remote Procedure Calling protocol. http://www.xmlrpc.com/.

[YGM01] Beverly Yang and Hector Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *Proceedings of the 27th International Conference of Very Large Databases (VLDB)*, pages 561–570, 2001.

[ZHS+04] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.